

Programming In Java

DECAP615

**Edited by
Balraj Kumar**



L OVELY
P ROFESSIONAL
U NIVERSITY



Programming In Java

**Edited By:
Balraj Kumar**

Title: PROGRAMMING IN JAVA

Author's Name: Harjinder Kaur

Published By : Lovely Professional University

Publisher Address: Lovely Professional University, Jalandhar Delhi GT road, Phagwara - 144411

Printer Detail: Lovely Professional University

Edition Detail: (I)

ISBN: 978-93-94068-30-8



Copyrights@ Lovely Professional University

CONTENT

Unit 1: Introduction	1
<i>Harjinder Kaur, Lovely Professional University</i>	
Unit 2: Arrays and Strings	23
<i>Harjinder Kaur, Lovely Professional University</i>	
Unit 3: Collection Framework	53
<i>Harjinder Kaur, Lovely Professional University</i>	
Unit 4: More on Collection Framework	71
<i>Harjinder Kaur, Lovely Professional University</i>	
Unit 5: Multithreading	86
<i>Harjinder Kaur, Lovely Professional University</i>	
Unit 6: More on Multithreading	101
<i>Harjinder Kaur, Lovely Professional University</i>	
Unit 7: Synchronization	115
<i>Harjinder Kaur, Lovely Professional University</i>	
Unit 8: Swings	127
<i>Harjinder Kaur, Lovely Professional University</i>	
Unit 9: More on Swings	145
<i>Harjinder Kaur, Lovely Professional University</i>	
Unit 10: Layouts	160
<i>Harjinder Kaur, Lovely Professional University</i>	
Unit 11: Managing Data using JDBC	179
<i>Harjinder Kaur, Lovely Professional University</i>	
Unit 12: More on JDBC	201
<i>Harjinder Kaur, Lovely Professional University</i>	
Unit 13: Network Programming	213
<i>Harjinder Kaur, Lovely Professional University</i>	
Unit 14: More on Network Programming	226
<i>Harjinder Kaur, Lovely Professional University</i>	

Unit 01: Introduction

CONTENTS

Objectives

Introduction

- 1.1 What is JAVA?
- 1.2 Java Platforms / Editions
- 1.3 Features of Java
- 1.4 Sample Program
- 1.5 Operators
- 1.6 Data Types
- 1.7 What is JDK
- 1.8 What is JVM?
- 1.9 What is JRE?
- 1.10 Wrapper Classes
- 1.11 Autoboxing and Unboxing
- 1.12 Nested Classes

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

Objectives

After this unit you will be able to:

- Understand the importance of Java programming.
- Learn the comparative analysis between java and C++.
- Identify the various features of Java.
- Know the different operators and data types of java.
- Learn the basics of JDK,JRE, and JVM
- Understand the working of JDK,JRE, and JVM
- Learn the basic concept and needsof wrapper classes.

Introduction

Java is an important object-oriented programming language that is used in the software industry today. Object-Oriented Programming is also known as OOP. Objects are the basic elements of object-oriented programming. OOPS (Object Oriented Programming System) is used to describe a computer application that comprises multiple objects connected. It is a type of programming language in which the programmers not only define the data type of a data structure (files, arrays, and so on) but also define the behavior of the data structure. Therefore, the data structure becomes an object, which includes both data and functions.

**Did you Know?**

Sun Microsystems originally wanted to name Java “OAK”. But it could not do so as that name was already taken by Oak Technologies. Other names that were suggested were “Silk and “DNA”. Ultimately, the name “Java” was selected because it gave the Web a “jolt”, and Sun intended to abstain from names that sounded very technical.

1.1 What is JAVA?

Java is a programming language and a platform. It is a high-level, robust, object-oriented, and secure programming language. It was developed by Sun Microsystems in the year 1995. James Gosling is known as the father of Java. Before Java, its name was Oak. Just like any other programming language, Java programs consist of some basic elements such as keywords, constants, variables, data types, operators, and expressions that help a programmer to create logical programs, and some important features such as platform independence which enables it to run on any computer platform.

1.2 Java Platforms / Editions

There are 4 platforms or editions of Java:

1. **Java SE (Java Standard Edition):** Java SE's API provides the core functionality of the Java programming language.
2. **Java EE (Java Enterprise Edition):** The Java EE platform provides an API and runtime environment for developing and running large-scale, multi-tiered, scalable, reliable, and secure network applications.
3. **Java ME (Java Micro Edition):** The Java ME platform provides an API and a small-footprint virtual machine for running Java programming language applications on small devices, like mobile phones.
4. **JavaFX:** JavaFX is a platform for creating rich internet applications using a lightweight user-interface API.

1.3 Features of Java

Let us understand Java better by understanding its important features. The following features of Java make it an important programming language:

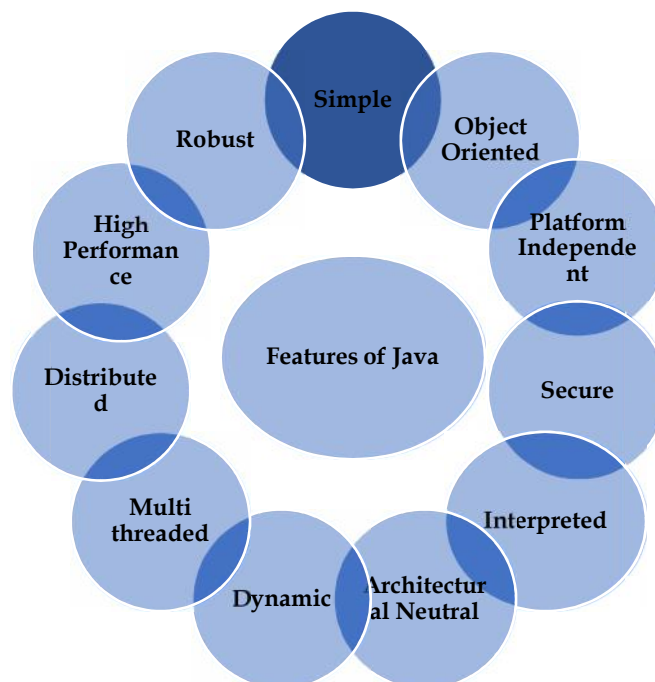


Figure 1: Features of Java Programming

Platform Independent: The write-once-run-anywhere approach towards programming is one of the key features of Java that makes it a powerful programming language. The programs written on one platform can run on any platform, irrespective of the hardware. But the hardware platform used to execute Java programs must have the Java Virtual Machine (JVM).

Simple: Various features make Java a simple language, which can be easily learned and effectively used. Java does not use pointers explicitly, thereby making it easy to write and debug programs. Java is capable of delivering a bug-free system due to its strong memory management. It also has an automatic memory allocation and de-allocation system in place.

Object-Oriented: To qualify as an object-oriented language, a language must exhibit four characteristics:

(a) **Inheritance:** It is the technique of creating new classes by making use of the behavior of the existing classes. This is done by extending the behavior of the existing classes just by adding additional features as required, thus bringing in the reusability of existing code.

(b) **Encapsulation:** It refers to the bundling of data along with the methods that act on that data.

(c) **Polymorphism:** Polymorphism, which means one name multiple forms, is the ability of a reference variable to change behavior according to the instance of the object that it holds.

(d) **Dynamic binding:** It is the method of providing maximum functionality to a program by resolving the type of object at runtime.

Robust: Java supports some features such as automatic garbage collection, strong memory allocation, powerful exception handling, and a type checking mechanism. The compiler checks the program for errors and the interpreter checks for any run time errors, thus preventing the system crash. These features make Java robust.

Distributed: The protocols like HTTP and FTP, which are extensively used over the Internet are developed using Java. Programmers who work on the Internet can call functions with the help of these protocols and can secure access to the files that are present on any remote machine on the Internet. This is made possible by writing codes on their local system itself.

Portable: The feature 'write-once run anywhere, anytime' makes Java portable, provided that the system has JVM. Java standardizes the data size, irrespective of the operating system or the processor. These features make Java a portable language.

Dynamic: A Java program also includes a significant amount of runtime information that is used to verify and resolve access to objects at runtime. This allows the code to link dynamically securely and appropriately.

Secure: Memory pointers are not explicitly used in Java. All programs in Java are run under a Java execution environment. Therefore, while downloading an applet program using the Internet, Java does not allow any virus or other harmful code to access the system as it confines it to the Java execution environment.

Performance: In Java, a program is compiled into an intermediate representation, which is called Java bytecode. This code can be executed on any system that has a JVM running on it. Earlier attempts to achieve cross-platform operability accomplished it at the cost of performance. Java bytecode is designed in such a manner that it is easy to directly translate the bytecode into the native machine code by using a just-in-time compiler. This helps in achieving high performance.

Multithreaded: The primary objective that led to the development of Java was to meet the real-world requirement of creating interactive and networked programs. To accomplish this, Java provides multithreaded programming, which permits a programmer to write programs that can do many things simultaneously.

Interpreted: Java programs can be directly run from the source code. The source code is read by the interpreter program and translated into computations. The source code generated is platform-independent. As an interpreted language, Java has an error debugging facility that can trace any error occurring in the program.

Architecture Neutral: These features of Java have made it a popular programming language. Java is also known as an architectural neutral language. In this era of networks, easy migration of applications to different computer systems having different hardware architectures and/or operating systems is necessary. The Java compiler generates an object file format that is architecture-neutral. This permits a Java application to execute anywhere on the network and many different processors, given the presence of the Java runtime system.

1.4 Sample Program

Let us understand the sections of the Java program in detail:

Package Section: This section consists of the first statement that is permitted in a Java file. This first statement is called the package statement. It gives the name of the package and provides information to the compiler that all the classes defined in the respective program are related to this package.

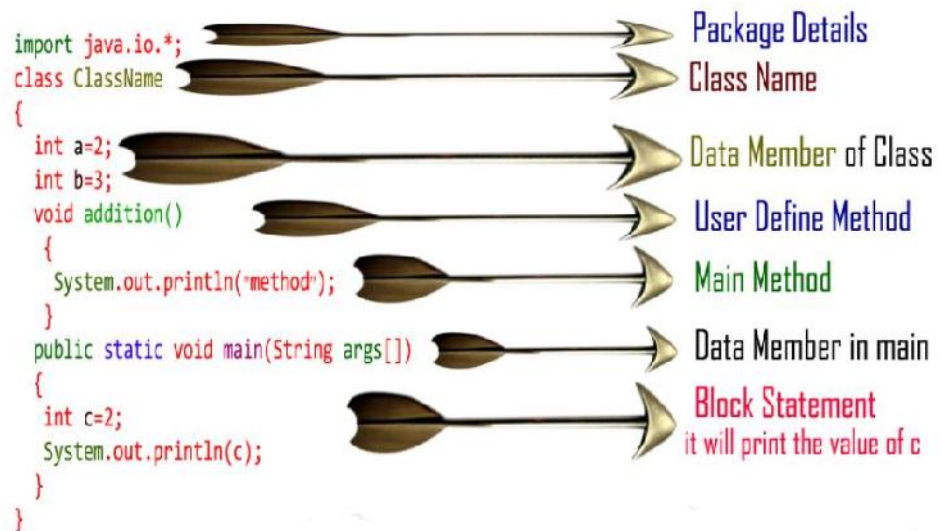


Figure 2: Program Structure



Example: package employee;

In this statement, the **package** is the keyword and the **employee** is the package name.

Import Statements' Section: This section consists of import statements, which can be used for accessing the classes that are related to other packages.



Example: package employee.salary;

In this statement, the interpreter is instructed to load the **salaried** class that is contained in the package **employee**.

Main Method Class Section: After understanding the basic elements, that is, keywords, variables, constants, and data types, a programmer can easily implement this structure while developing programs in Java.

This section consists of the definition of the main method in the program. The main method enables the creation of objects of different classes and the communication between these objects.

Compiling, Interpreting, and Running the Program

It is necessary to convert the program into a format that can be understood by the JVM before any computer with a JVM can interpret and run the program. The process of compiling a Java program involves obtaining the programmer-readable text in the program file (source code) and translating it to bytecode. Bytecode is nothing but the platform-independent instructions for the JVM.



Caution: In Java, the class name and the file name must be the same. Also, since Java is case-sensitive, one should always provide the same case letters for both file name and class name.

After compiling and interpreting, a program is run. To run a program, follow the instructions given below:

1. Go to the command prompt.
2. Select the drive (and the folder) where the program file is saved.

3. Compile, Interpret and Run the program.



Example: Type the following program and save it in a file with the name "FirstProgram.java"

```
classFirstProgram
{
public static void main(String args[ ])
{
System.out.println("My first program in Java");
}
}
```

The process to compile and run the above-given example is described in the following steps

1. In the command prompt, type the directory name, and the file name.

Example: c:\jdk\programs>javac FirstProgram.java

Example C:\jdk\programs > java FirstProgram



Task: Write a simple Java program to print "Java".

1.5 Operators

In real-time, programs are required to perform a lot more than just simple input and output operations. All computer languages provide tools to perform some predefined operations. To facilitate this, there are different types of operators provided in Java. These are arithmetic operators, relational operators, logical operators, and conditional operators. These operators are assigned a certain kind of precedence so that the compiler evaluates the operations based on their precedence. This helps when there are multiple operators in one statement. There are many types of operators in Java which are given below:

Unary Operator: The Java unary operators require only one operand. Unary operators are used to performing various operations i.e.:

- incrementing/decrementing a value by one
- negating an expression
- inverting the value of a boolean

Arithmetic Operator: Java arithmetic operators are used to perform addition, subtraction, multiplication, and division. They act as basic mathematical operations.

Shift Operator: The shift operator works on bits and performs shift operations to move bits either leftwards or rightwards.

Relational Operator: The Java Relational operators compare operands and determine the relationship between them.

Bitwise Operator: Bitwise operators are used to performing the manipulation of individual bits of a number. They can be used with any of the integral types (char, short, int, etc).

Logical Operator: The Java Logical Operators work on the Boolean operand. It's also called Boolean logical operators. It operates on two Boolean values, which return Boolean values as a result.

Ternary Operator: The Java Conditional Operator selects one of two expressions for evaluation, which is based on the value of the first operands. It is also called the ternary operator because it takes three arguments.

Assignment Operator: The Java Assignment Operators are used when you want to assign a value to the expression. The assignment operator is denoted by the single equal sign =.

Table 1: Operator Precedence

Programming in JAVA

Operator Type	Category	Precedence
Unary	postfix	expr++ expr--
	prefix	++expr --expr
Arithmetic	multiplicative	* / %
	additive	+ -
Shift	shift	<<>>>>
Relational	comparison	<> <= >= instanceof
	equality	== !=
Bitwise	bitwise AND	&
	bitwise exclusive OR	^
	bitwise inclusive OR	
Logical	logical AND	&&
	logical OR	
Ternary	ternary	? :
Assignment	assignment	= += -= *= /= %= &= ^= = <<= >>= >>>=

1.6 Data Types

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

- **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float, and double.
- **Non-primitive data types:** The non-primitive data types include Classes, Interfaces, and Arrays.

Table 2: Data Type with default values

Data Type	Default Value	Default size
boolean	false	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte

int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

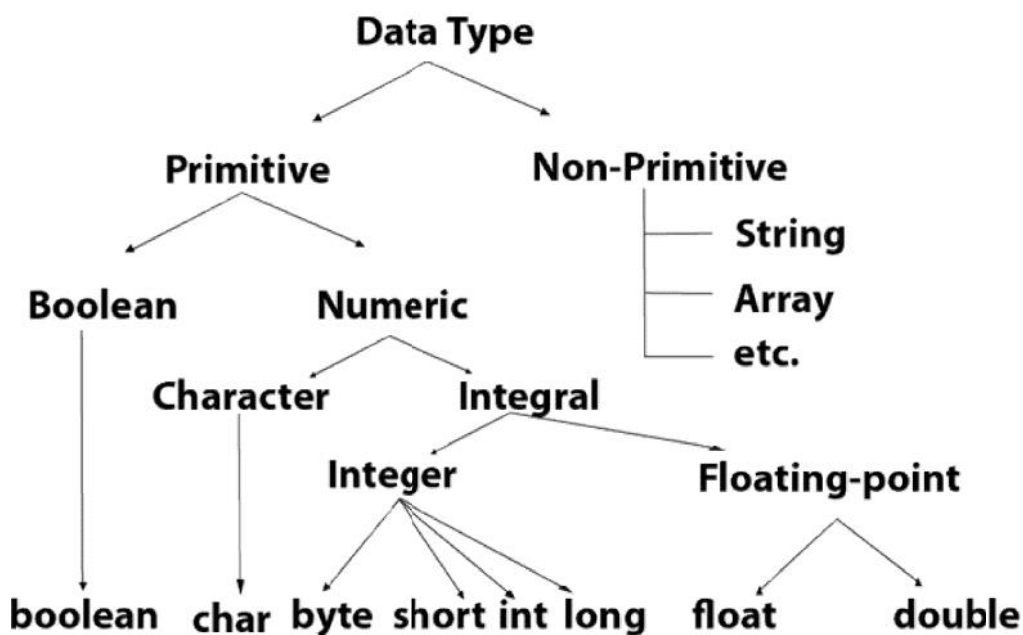


Figure 3: Different data types

Boolean Data Type: The Boolean data type is used to store only two possible values:

true

false.



Example: Boolean one = false.

Byte Data Type: It is an 8-bit signed two's complement integer. Its value range lies between -128 to 127 (inclusive). Its minimum value is -128 and its maximum value is 127. Its default value is 0. It saves space because a byte is 4 times smaller than an integer.



Example: byte a = 10, byte b = -20

Short Data Type: The short data type is a 16-bit signed two's complement integer. Its value range lies between -32,768 to 32,767 (inclusive). A short data type is 2 times smaller than an integer.



Example: short s = 10000, short r = -5000

int data type: The int data type is a 32-bit signed two's complement integer. Its value range lies between -2,147,483,648 (-2^{31}) to 2,147,483,647 ($2^{31}-1$) (inclusive).



Example: int a = 100000, int b = -200000

Float Data Type: The float data type is a single-precision 32-bit IEEE 754 floating-point. Its value range is unlimited. The float data type should never be used for precise values, such as currency. Its default value is 0.0F.



Example: float f1 = 234.5f

Double Data Type: The double data type is a double-precision 64-bit IEEE 754 floating-point. Its value range is unlimited. The double data type is generally used for decimal values just like float.



Example: double d1 = 12.3

Char Data Type: The char data type is a single 16-bit Unicode character. Its value-range lies between '\u0000' (or 0) to '\uffff' (or 65,535 inclusive). The char data type is used to store characters.



Example: char letterA = 'A'

1.7 What is JDK

JDK stands for Java development kit. It is a cross-platformed software development environment that offers a collection of tools and libraries necessary for developing Java-based software applications and applets. It is a core package used in Java, along with the JVM (Java Virtual Machine) and the JRE (Java Runtime Environment). It is used to build and develop the java program. It internally contains JRE. It contains the compiler and debugger.

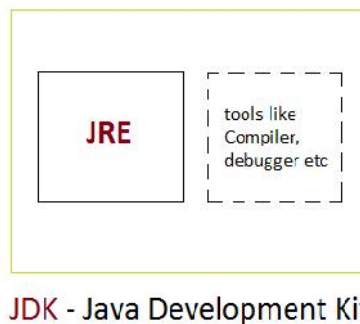


Figure 4:JDK

It contains all the related sets of libraries and files to build and compile the java program. Without JDK we can't build any java program. The following are the basic tools that become the foundation of the Java Development Kit.

Javac: Javac is the compiler for the Java programming language. It is used to compile the .java file. It creates a class file that can be run by using the java command.



Example: c:\javac TestFile.java

Java: When a class file has been created, the java command can be used to run the Java program.



Example: c:\java TestFile.class

Javadoc: Javadoc is an API documentation generator for the Java language, which generates documentation in HTML format from Java source code.

Appletviewer: Appletviewer runs and debugs applets without a web browser, it is a standalone command-line program to run Java applets.

Jar: The jar is (manage Java archive) a package file format that contains class, text, images, and sound files for a Java application or applet gathered into a single compressed file.

1.8 What is JVM?

It is a specification where the working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Oracle and other companies.

An implementation: Its implementation is known as JRE (Java Runtime Environment).

Runtime Instance Whenever you write a java command on the command prompt to run the java class, an instance of JVM is created.

Mostly in other Programming Languages, the compiler produces code for a particular system but the Java compiler produces Bytecode for a Java Virtual Machine. When we compile a Java program, then bytecode is generated that can be used to run on any platform. It is the medium that compiles Java code to bytecode which gets interpreted on a different machine and hence it makes it Platform/Operating system independent.

- Java is called platform independent because of Java Virtual Machine.
- JVM is the main component of Java architecture.
- As different computers with different operating systems have their JVM, when we submit a .class file to any operating system, JVM interprets the bytecode into machine-level language.
- A program of JVM is written in C Programming Language, and JVM is Operating System dependent.
- JVM is responsible for allocating the necessary memory needed and also responsible for deallocating the memory space not required by the Java program.

Working of JVM

The JVM performs the following operation:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

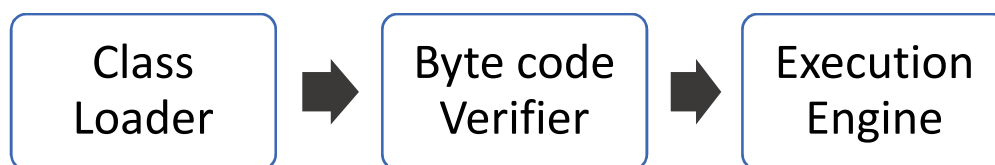


Figure 5: JVM Working

JVM Architecture

1) Classloader

The class loader is a subsystem used for loading class files. It performs three major functions viz. Loading, Linking, and Initialization.

2) Method Area

JVM Method Area stores class structures like metadata, the constant runtime pool, and the code for methods.

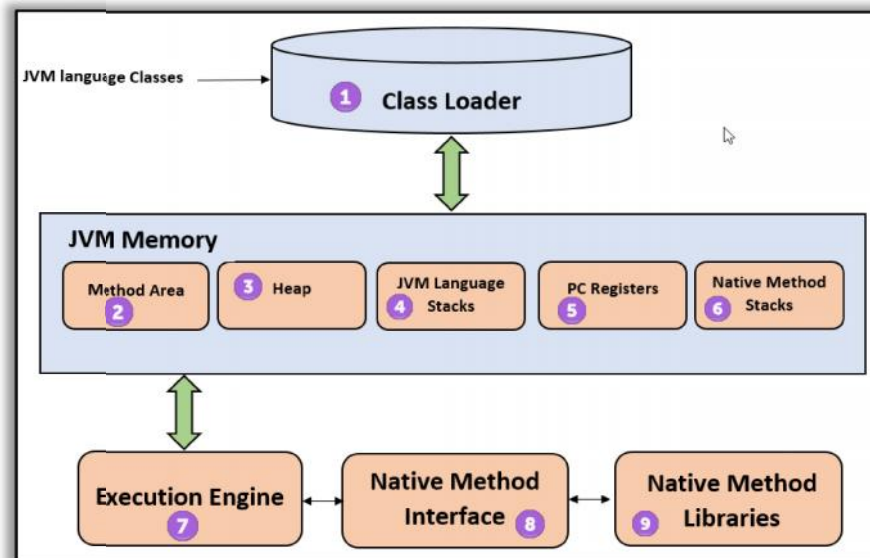


Figure 6: JVM Architecture

3) Heap

All the Objects, their related instance variables, and arrays are stored in the heap. This memory is common and shared across multiple threads.

4) JVM language Stacks

Java language Stacks store local variables, and it's partial results. Each thread has its own JVM stack, created simultaneously as the thread is created. A new frame is created whenever a method is invoked, and it is deleted when the method invocation process is complete.

5) PC Registers

PC register stores the address of the Java virtual machine instruction which is currently executing. In Java, each thread has its separate PC register.

6) Native Method Stacks

Native method stacks hold the instruction of native code depending on the native library. It is written in another language instead of Java.

7) Execution Engine

It is a type of software used to test hardware, software, or complete systems. The test execution engine never carries any information about the tested product.

8) Native Method interface

The Native Method Interface is a programming framework that Java code which is running in a JVM to call by libraries and native applications.

9) Native Method Libraries

Native Libraries is a collection of the Native Libraries(C, C++) that are needed by the Execution Engine.

1.9 What is JRE?

A Java runtime environment (JRE) is a set of components to create and run a Java application. A JRE is part of a Java development kit (JDK). A JRE is made up of :

- ✓ Java virtual machine (JVM)
- ✓ Java class libraries
- ✓ Java class loader.

Once we write any source code, we have to save it with the .java extension. Next, we compile our program using the “javac” command in the command prompt. It translates the code to the platform-independent bytecode. When we compile our program it will generate a .class that has the bytecode for the equivalent java file. And that bytecode is executed on any platform having the JRE.

Working of JRE

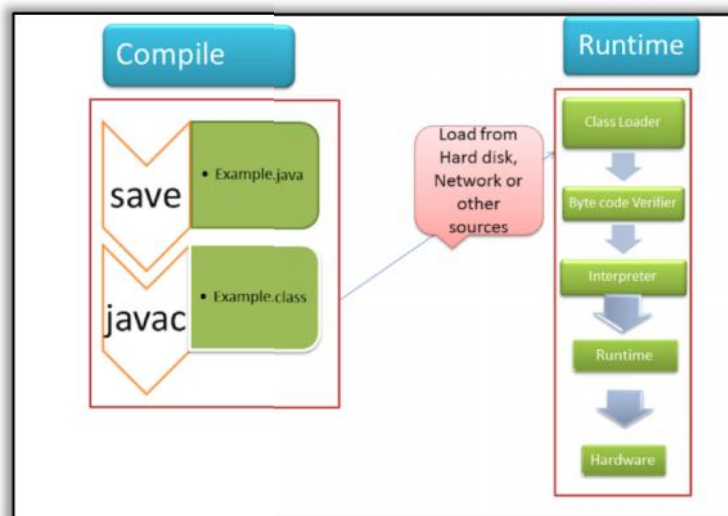


Figure 7: Working of JRE

The workflow is explained below. ClassLoader (It is also called the **Primordial ClassLoader**.) It loads the various classes dynamically in the JVM (Java Virtual Machine), which are essential for executing the program. After JVM started following three class loader are used:

Bootstrap class loader: A Bootstrap Classloader is a Machine code that kickstarts the operation when the JVM calls it. Its job is to load the first pure Java ClassLoader. Bootstrap ClassLoader loads classes from the location *rt.jar*.

Extension class loader: The Extension ClassLoader is a child of Bootstrap ClassLoader and loads the extensions of core java classes from the respective JDK Extension library.

System class loader: An Application ClassLoader is also known as a System ClassLoader. It loads the Application type classes found in the environment variable *CLASSPATH*, *-classpath*, or *-cp command-line option*. The Application ClassLoader is a child class of Extension ClassLoader.

Byte code verifier: The **bytecode verifier** acts as a sort of gatekeeper: it ensures that code passed to the **Java** interpreter is in a fit state to be executed and can run without fear of breaking the **Java** interpreter. Imported code is not allowed to execute by any means until after it has passed the **verifier's** tests. It verifies the byte code during the run time so that the code doesn't make any disturbance for the interpreter. The codes are interpreted, only if it passes the tests of the bytecode verifier that check the formatting and for illegal code.

Interpreter: When the class is loaded and the code gets verified, then it reads the assembly code line by line and processes the following two functions: It executes the Byte Code. It makes appropriate calls to the integrated hardware.

Table 3: Difference Between JDK, JRE, and JVM

JDK	JRE	JVM
The full form of JDK is Java Development Kit.	The full form of JRE is Java Runtime Environment.	The full form of JVM is Java Virtual Machine.
JDK is platform-dependent.	JRE is also platform-dependent.	JVM is platform-independent.
JDK is a software development kit to develop	It is a software bundle that provides Java class libraries	JVM executes Java byte code and provides an environment

Programming in JAVA

applications in Java.	with the necessary components to run Java code.	for executing it.
It is the superset of JRE	It is the subset of JDK.	JVM is a subset of JRE.
JDK comes with the installer.	JRE only contains an environment to execute source code.	JVM bundled in both software JDK and JRE.
It contains tools for developing, debugging, and monitoring java code.	It contains class libraries and other supporting files that JVM requires to execute the program.	Software development tools are not included in JVM.
It contains tools for developing, debugging, and monitoring java code.	It contains class libraries and other supporting files that JVM requires to execute the program.	Software development tools are not included in JVM.

1.10 Wrapper Classes

Java uses primitive types, such as int, char, double to hold the basic data types supported by the language. Sometimes it is required to create an object representation of these primitive types. These are collection classes that deal only with such objects. One needs to wrap the primitive type in a class. A wrapper class is a class that envelops the value of every primitive data type. This class represents primitive data types in their equivalent class instances. The **wrapper class in Java** provides the mechanism to convert *primitive into object and object into primitive*. **Autoboxing** and **unboxing** features convert primitives into objects and objects into primitives automatically. The automatic conversion of a primitive into an object is known as autoboxing and vice-versa unboxing.

These classes encapsulate or wrap, the primitive types within a class. Thus, they are commonly referred to as type wrappers. Type wrapper is a class that encapsulates a primitive type within an object. The wrapper types are Byte, Short, Integer, Long, Character, Boolean, Double, Float.

Use of Wrapper classes in Java

Java is an object-oriented programming language, so we need to deal with objects many times like in Collections, Serialization, Synchronization, etc. Let us see the different scenarios, where we need to use the wrapper classes.

- **Change the value in Method:** Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value into an object, it will change the original value.
- **Serialization:** We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it into objects through the wrapper classes.
- **Synchronization:** Java synchronization works with objects in Multithreading.
- **java.util package:** The java.util package provides the utility classes to deal with objects.
- **Collection Framework:** Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

The eight classes of the *java.lang* packages are known as wrapper classes in Java.

Table 4: primitive and its corresponding Wrapper class

Primitive Type	Wrapper class
boolean	Boolean

char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Difference b/w Primitive Data Type and Object of a Wrapper Class

The following two statements illustrate the difference between a primitive data type and an object of a wrapper class:

```
int x = 25;
```

```
Integer y = new Integer(33);
```

The first statement declares an int variable named x and initializes it with the value 25. The second statement instantiates an Integer object. The object is initialized with the value 33 and a reference to the object is assigned to the object variable y. X and y differ by more than their values:

- x is a variable that holds a value;
- y is an object variable that holds a reference to an object.

Need of Wrapper Classes

- Wrapper classes are used to be able to use the primitive data types as objects.
- Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.
- An object is needed to support synchronization in multithreading.
- The classes in java.util package handle only objects and hence wrapper classes help in this case also.

Boxing and Unboxing

The wrapping is done by the compiler. If we use a primitive where an object is expected, the compiler boxes the primitive in its wrapper class. Similarly, if we use a number object when a primitive is expected, the compiler unboxes the object.



Example of boxing and unboxing

```
Integer x, y;
```

```
x = 12; y = 15; //boxing
```

```
System.out.println(x+y); //Unboxing
```

- When x and y are assigned integer values, the compiler boxes the integers because x and y are integer objects.
- In the println() statement, x and y are unboxed so that they can be added as integers.

Features of Wrapper Classes

Wrapper classes have the following features:

1. Methods defined within wrapper classes are **static**.

Programming in JAVA

2. Wrapper classes are immutable, that is, we cannot change any value after assigning it to a wrapper

class.



Example : In **java.lang.Integer**, **java.lang** is the name of the wrapper library, and **Integer** is the wrapper name.

Numeric Wrapper Classes

All of the numeric wrapper classes are subclasses of the abstract class **Number**. Short, Integer, Double, and Long implement Comparable interface.

Nested Classes

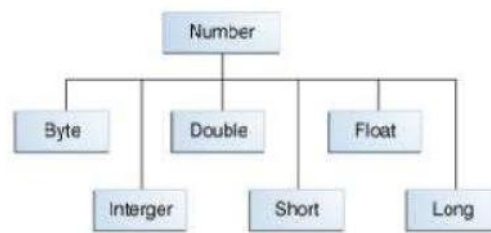


Figure 8: Types of Numeric wrapper class

Features of Numeric Wrapper Classes

- ✓ All the numeric wrapper classes provide a method to convert a numeric *string* into a *primitive value*.

Syntax: `public static type parseType (String Number)`

- `parseInt()`
- `parseFloat()`
- `parseDouble()`
- `parseLong()`

- ✓ All the wrapper classes provide a static method **toString** to provide the *string representation* of the *primitive values*.

Syntax: `public static String toString (type value)`

Example: `public static String toString (int a)`

- ✓ All numeric wrapper classes have a static method **valueOf**, which is used to *create a new object initialized to the value* represented by the specified string.

Syntax: `public static DataType valueOf (String s)`



Example:

- `Integer i = Integer.valueOf ("135");`
- `Double d = Double.valueOf ("13.5");`

Character Class

Character is a wrapper around a `char`. The constructor for Character is :

Character(char ch)

- Here, `ch` specifies the character that will be wrapped by the Character object being created.
- To obtain the `char` value contained in a Character object, call `charValue()`, shown here:

```
char charValue();
```

- It returns the encapsulated character.

Boolean Class

Boolean is a wrapper around boolean values. It defines these constructors:

- Boolean(boolean boolValue)
- Boolean(String boolString)

In the first version, boolValue must be either true or false. In the second version, if boolString contains the string "true" (in uppercase or lowercase), then the new Boolean object will be true. Otherwise, it will be false. To obtain a boolean value from a Boolean object, use booleanValue(), shown here:

```
boolean booleanValue()
```

It returns the boolean equivalent of the invoking object.

1.11 Autoboxing and Unboxing

The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing.



Example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.

```
//Program for Autoboxing
public class AutoBoxingTest {
    public static void main(String args[]) {
        int num = 10; // int primitive
        Integer obj = Integer.valueOf(num); // creating a wrapper class object
        System.out.println(num + " " + obj);
    }
}
```

Output: 10 10

The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing.

```
//Program for Unboxing
public class UnboxingTest {
    public static void main(String args[]) {
        Integer obj = new Integer(10); // Creating Wrapper class object
        int num = obj.intValue(); // Converting the wrapper object to primitive datatype
        System.out.println(num + " " + obj);
    }
}
```

Output: 10 10

1.12 Nested Classes

The Java programming language allows you to define a class within another class. Such a class is called a *nested class*.

Syntax:

```
class OuterClass {
```

```
...  
classNestedClass {  
...  
}}
```

Properties of Nested Class

- The scope of a nested class is bounded by the scope of its enclosing class.
- A nested class has access to the members, including private members, of the class in which it is nested.
- However, the reverse is not true i.e., the enclosing class does not have access to the members of the nested class.
- A nested class is also a member of its enclosing class.
- As a member of its enclosing class, a nested class can be declared as private, public, protected, or package-private.

Why Use Nested Classes?

The following are the reasons for using nested classes:

- **Logical grouping of classes**—If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together.
- **Increased encapsulation**—Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.
- **More readable, maintainable code**—Nesting small classes within top-level classes places the code closer to where it is used.

Categories of Nested Classes

Nested classes are divided into two categories:

1. Static

2. Non-static.

- Nested classes that are declared **static** are called *static nested classes*.
- **Non-static** nested classes are called *inner classes*.

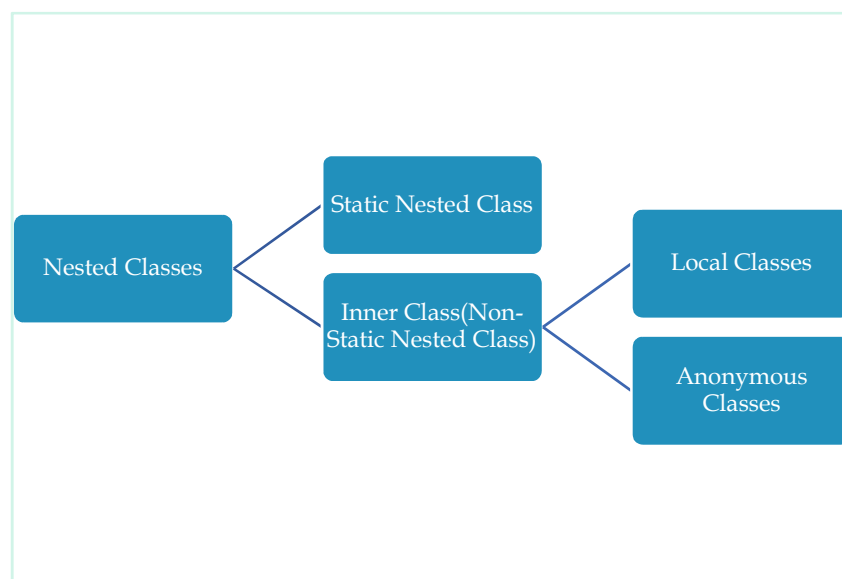


Figure 9: Categories of Nested Classes

Static Nested Class

A static class i.e. created inside a class is called a static nested class in java. It cannot access non-static data members and methods. It can be accessed by outer class name. It can access static data members of the outer class including private. The static nested class cannot access non-static (instance) data members or methods. A static nested class is associated with its outer class similar to class methods and variables. A static nested class cannot refer directly to instance variables or methods defined in its enclosing class. It can use them only through an object reference.

Program with static method

If you have the static member inside static nested class, you don't need to create instance of static nested class.

```
class TestOuter2{
    static int data=30;
    static class Inner{
        static void msg(){System.out.println("data is "+data);}
    }
    public static void main(String args[]){
        TestOuter2.Inner.msg();//no need to create the instance of static nested class    }
    }
}
```

Table 5: Types of Inner Class

Type	Description
Member Inner Class	A class created within class and outside method.
Anonymous Inner Class	A class created for implementing an interface or extending class. Its name is decided by the java compiler.
Local Inner Class	A class created within a method.
Static Nested Class	A static class is created within the class.
Nested Interface	An interface created within class or interface.

Java Member inner class

A non-static class that is created inside a class but outside a method is called member inner class.

Syntax:

```
class Outer{
    //code
    class Inner{
        //code
    }
}
```

In this program, we are creating msg() method in member inner class that is accessing the private data member of outer class.

```
class TestMemberOuter1{
    private int data=30;
```

Programming in JAVA

```

class Inner{
    void msg(){System.out.println("data is "+data);} //
}

public static void main(String args[]){
    TestMemberOuter1 obj=new TestMemberOuter1();
    TestMemberOuter1.Inner in=obj.new Inner();
}

```

Java Anonymous inner class

A class that has no name is known as an anonymous inner class in java. It should be used if you have to override a method of class or interface. Java Anonymous inner class can be created in two ways:

- Class (may be abstract or concrete).
- Interface

Program Using Class

```

abstract class Person{
    abstract void eat();
}

class TestAnonymousInner{
    public static void main(String args[]){
        Person p=new Person(){
            void eat(){System.out.println("nice fruits");}
        };
        p.eat();
    }
}

```

Program Using Interface

```

interface Eatable{
    void eat();
}

class TestAnonymousInner1{
    public static void main(String args[]){
        Eatable e=new Eatable(){
            public void eat(){System.out.println("nice fruits");}
        };
        e.eat();
    }
}

```

Local inner class

A class created inside a method is called the local inner class in java. If you want to invoke the methods of the local inner class, you must instantiate this class inside the method.

Program:

```

public class localInner1{
    private int data=30;//instance variable
}

```

```

void display(){
    class Local{
        void msg()
    {
        System.out.println(data);}
    }
    Local l=new Local();
    l.msg();
}

public static void main(String args[]){
    localInner1 obj=new localInner1();
    obj.display();
}
}

```

Rules for Local Inner class

- A local inner class cannot be invoked from outside the method.
- A local inner class cannot access non-final local variables till JDK 1.7. Since JDK 1.8, it is possible to access the non-final local variable in the local inner class.

Table 6: Comparison between Inner and Static Nested Class

inner class	Static nested class
The inner class object is always associated with the outer class object.	A static nested class object is not associated with the outer class object.
Inside the inner class, static members can't be declared.	Inside the static nested class, static members can be declared.
As the main() method can't be declared, the regular inner class can't be invoked directly from the command prompt.	As the main() method can be declared, the static nested class can be invoked directly from the command prompt.
Both static and non-static members of an outer class can be accessed directly.	Only a static member of an outer class can be accessed directly.

Summary

Java was developed by Sun Microsystems initially to offer solutions for household appliances. But, finally, it evolved as a fully functional programming language.

Java has many features such as platformindependence, simplicity, object-oriented capability, portability, and so on that differentiates it from other programming languages and makes it important.

The structure of a Java program consists of a comment lines' section, package section, import statements' section, interface statements' section, class definitions' section, and main method class section.

All Java keywords have predefined meanings and distinct functions. They must be written in lowercase letters.

Every variable has a data type associated with it, which indicates the type and the size of values that can be stored in that variable.

Programming in JAVA

Data types are of two types – primitive and non-primitive data types. Primitive data types are inbuilt

in any programming language and are used for categorically storing information or data that may or may not be interchangeable. Non-primitive data types are defined by the user.

In Java, operators such as arithmetic operators, relational operators, logical operators, and conditional operators are used to perform some predefined operations.

Keywords

boolean: Java boolean keyword is used to declare a variable as a boolean type. It can hold True and False values only.

char: Java char keyword is used to declare a variable that can hold unsigned 16-bit Unicode characters

class: Java class keyword is used to declare a class.

float: Java float keyword is used to declare a variable that can hold a 32-bit floating-point number.

Object: Object means a real-world entity such as a pen, chair, table, computer, watch, etc.

Initialize: Allocate an initial value to a java program.

Instance Variable: A variable that is relevant to a single instance (an object belonging to a class is an instance of that class) of a class.

Primitive Data Types: Special group of data types that represents a single value such as numbers, characters, or Boolean values.

Wrapper Library: Library of wrappers of all the primitive data types.

OOP: Object-Oriented Programming.

Operators' Precedence: Order in which the operators are executed during expression evaluation.

Self Assessment

- 1 _____ is the mechanism that enables us to combine the information and provide abstraction.
- 2 All keywords must be written in _____ letters.
- 3 The keyword _____ is used to declare the constants.
- 4 While comparing primitive variables, the actual values are compared, but while comparing reference variables, the _____ are compared.
- 5 _____ data types are used to categorically store information or data that may or may not be interchangeable, and that is built-in.
- 6 When variables, constants, and operators that are arranged according to the syntax of the programming language are combined, an _____ is formed.
- 7 Which of the following option leads to the portability and security of Java?
 - A. Bytecode is executed by JVM
 - B. The applet makes the Java code secure and portable
 - C. Use of exception handling
 - D. Dynamic binding between objects
- 8 Which concept of Java is achieved by combining methods and attribute into a class?
 - A. Encapsulation
 - B. Inheritance

- C. Polymorphism
- D. Abstraction

9 Which is the Logical operator in Java that works with a Single Operand?

- A. Logical AND
- B. Logical OR
- C. Logical Exclusive OR
- D. Logical NOT

10 What is the output of a Logical OR (|) operation if one of the inputs/operands is true?

- A. false
- B. true
- C. true or false
- D. None of the above

11 What does a Data Type in Java refer to?

- A. The place where data is stored
- B. The technique of how data is retrieved
- C. The type or variety of data being handled for reading and writing
- D. None of the above

12 Which among the following is not a valid Data Type in Java?

- A. long
- B. bool
- C. double
- D. float

13 Which among the following best describes a nested class?

- A. Class inside a class
- B. Class inside a function
- C. Class inside a package
- D. Class inside a structure

14 Which is true about an anonymous inner class?

- A. It can extend exactly one class and implement exactly one interface.
- B. It can extend exactly one class and can implement multiple interfaces.
- C. It can extend exactly one class or implement exactly one interface.
- D. It can implement multiple interfaces regardless of whether it also extends a class.

15 The nested class can be declared _____

- A. Public
- B. Private
- C. Protected

D. Public, Protected, Private or Package private

Answers for Self Assessment

- | | | | | |
|--------------------------|--------------|----------|--------------|--------------|
| 1. Encapsulation | 2. Lowercase | 3. Final | 4. Addresses | 5. Primitive |
| 6. Arithmetic Expression | 7. A | 8. A | 9. D | 10. B |
| 11. C | 12. B | 13. A | 14. C | 15. D |

Review Questions

1. Analyze different features of Java, which has made Java an important programming language.
2. "Structure of a Java program consists of six sections." Justify.
3. "Declaration of a variable involves specifying the data type of the variable." Discuss.
4. "Primitive variables and reference variables are different from each other." Justify.
5. "In every Java program, different types of data are used." Elaborate.
6. 11. "Increment and decrement operators are used to increment or decrement values." Elaborate.
7. 12. "To run a program in Java, the user can use a command prompt." Discuss.
8. "Nested classes enhance the readability and maintainability of code". Comment.
9. "A wrapper class is a class that envelops the value of every primitive data type." Discuss.



Further Readings

Balagurusamy E. Programming with Java_A Primer 3e. New Delhi
 Schildt. H. Java 2 The Complete Reference, 5th ed. New York: McGraw-Hill/Osborne.
 King, K. N. (2000). Java programming: from the beginning. WW Norton & Co., Inc.
 Sanchez, J., & Canton, M. P. (2002). Java programming for engineers. CRC Press



Web Links

www.roseindia.net/java/java-introduction/java-features.shtml
www.tech-faq.com/java-data-types.html - United States
<http://java.sun.com/j2se/press/bama.html>
http://docstore.mik.ua/orelly/java/javanut/ch03_01.htm
<http://www.java-tips.org/java-se-tips/java.lang/what-is-a-final-class.html>
<https://www.javatpoint.com/java-basics>
<https://www.geeksforgeeks.org/java-programming-basics/>

Unit 02: Arrays and Strings

CONTENTS

Objectives

Introduction

2.1 Arrays

2.2 Types of Array

2.3 Two Dimensional Array

2.4 Strings

2.5 Different Ways of Creating Strings

2.6 String Methods

2.7 String Array

2.8 StringBuffer Class

2.9 StringBuilder Class

2.10 Access Specifiers

2.11 Inheritance

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

Objectives

After this unit you will be able to:

- Learn the basic concept of Arrays.
- Understand the different types of Arrays.
- Implement the difference in one-dimensional and multi-dimensional Arrays.
- Learn the basic concept of Strings.
- Understand the ways of creating Strings.
- Use different String Functions.
- Implement the various methods of StringBuffer and StringBuilder Class.
- Differentiate between StringBuffer and StringBuilder Class.
- Understand the different types of access specifiers.
- Implementation of different types of inheritance.

Introduction

Every programming language has some important concepts that make programming more easy and effective. Arrays and strings are such important concepts and are available in Java. The concept of arrays is used in a program when similar types of values for a large number of data items

Programming in JAVA

are to be stored by the user. Arrays can be used for many applications such as performing calculations on statistical data and representing the state of a game.



Example: To store the salaries of all the employees of a company, a declaration of thousands of variables will be required. Also, the name of every variable must be unique. In such situations, arrays can be used for simple and easy storage.

An array is an object that contains a fixed number of values of a single data type and an array object consists of several variables. These variables are called array elements. In Java, arrays can be created dynamically. The number of variables can be even zero; in such cases, the array is said to be empty. Once the array is declared, continuous space is allocated in the memory for storing these variables. The elements in the array are accessed by referring to their index value.

We can define a string as a collection of characters. Java handles character strings by using two final classes, namely, `String` class and `StringBuffer` class. The `String` class is used to implement character strings that are immutable and read-only after the creation and initialization of the string. The `StringBuffer` class is used to implement dynamic character strings.

2.1 Arrays

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value. It is an object which contains elements of a similar data type. The elements of an array are stored in a contiguous memory location. We can store only a fixed set of elements in a Java array. Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on the 1st index, and so on.

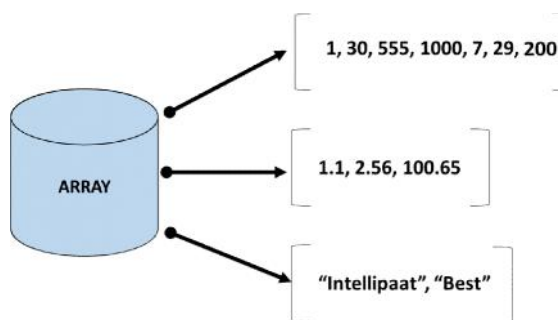


Figure 1: Storage of different elements in an array



Example: `int salary[] = new int[50];`

In this example, an array is created to store the salaries of 50 employees, with an array size of 50. This array is given a common name for the salaries of these 50 employees, that is, salary.

In Java, all arrays are dynamically allocated. Since arrays are objects in Java, we can find their length using the object property `length`. This is different from C/C++ where we find length using `sizeof`.

Declaring an Array

As variables are declared before their usage in the program, so are arrays. To declare an array, define the variable type with **square brackets** after the type, followed by the name of the identifier.:

`type var-name[];`

OR

`type[] var-name;`



Example: `int[] array = new int [5];` // creates an array of integers.

In this example, an array named `array` is created, which stores 5 integer values in it.

The size of an array must be specified by `int` or `short` value and not `long`. The direct superclass of an array type is `Object`. Every array type implements

the interfaces Cloneable and java.io.Serializable. Java array can also be used as a static field, a local variable, or a method parameter.

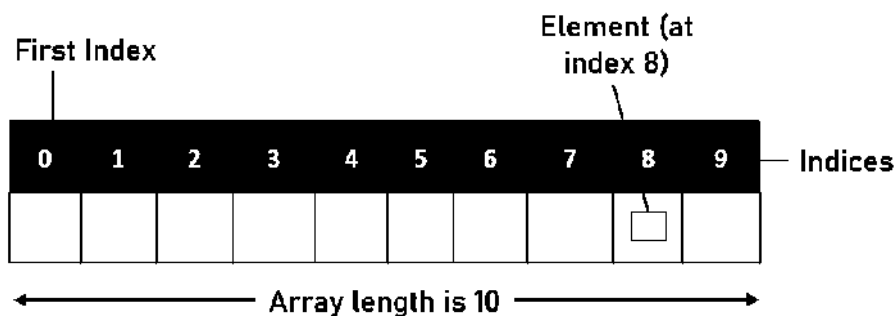


Figure 2: Representation of Array elements in memory

Creating an Array

We create arrays after declaring them. We have to specify the length of the array by using a constructor that is called for the creation of a particular object. The **new** keyword is used for creating an array, which is followed by the number of elements to be contained in that array (in square brackets). In any program, the array length is fixed when the array is created. We know that an array is used to represent a group of entities with the same data type in adjacent memory locations and these data items are given a common name.



Example: `students[5];`

Here, the name of the array is `students`, which is of size (index value) 5. This complete group of values is known as an array, and the individual values are called the array elements.

Initializing an Array

After the creation of an array, we need to initialize it and give a value. This means memory is allocated to that array, and this is done with the help of a new operator, or simply at the time of declaration. If an array of integers is not initialized in the program, it will start the index numbers from 0 as it is automatically initialized to 0 at the time of creation.

Accessing an Array Element

Once the array is created and the elements are allocated in the array, a specific element of the array can be accessed by specifying its index in the square brackets. An index number is assigned to each element of the array, which allows the program and the programmer to access the individual values of the array. Index numbers are integers. These index numbers always start with zero and progress sequentially till the end of the array.



Example: `int[] arrayint = {1, 2, 3, 4, 5};`

`arrayint[1] = 2;`

As the array index always starts from zero, the value at index 1 of the array `arrayint` is 2.

2.2 Types of Array

Java supports different types of arrays such as one-dimensional arrays and two-dimensional arrays.

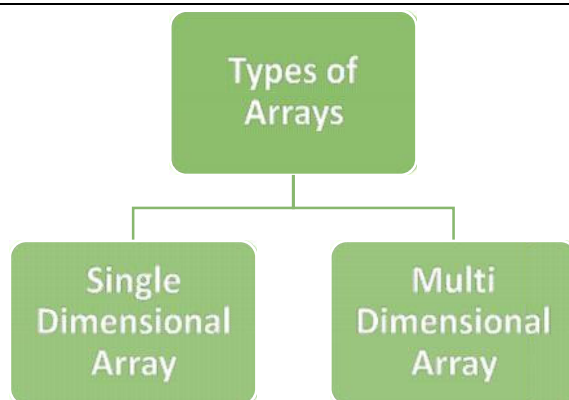


Figure 3:Types of Array

One-dimensional (1D) Arrays

The 1D array can be defined as a variables' list containing the data of a similar type.

Syntax:

```
type var-name[];
```

OR

```
type[] var-name;
```

An array declaration has two components: the type and the name. *type* declares the element type of the array. The element type determines the data type of each element that comprises the array. Like an array of integers, we can also create an array of other primitive data types like char, float, double, etc., or user-defined data types (objects of a class). Thus, the element type for the array determines what type of data the array will hold.

Instantiating an Array

When an array is declared, only a reference of an array is created. The general form of new as it applies to one-dimensional arrays appears as follows:

```
var-name = new type [size];
```

Here, *type* specifies the type of data being allocated, *size* specifies the number of elements in the array, and *var-name* is the name of the array variable that is linked to the array. That is, to use *new* to allocate an array, you must specify the type and number of elements to allocate.



Example: `int intArray[]; //declaring array`

```
intArray = new int[20]; // allocating memory to array
```

OR

```
int[] intArray = new int[20]; // combining both statements in one
```



Did you know? We cannot resize an array after it has been declared.

Accessing Array Elements using for Loop

- Each element in the array is accessed via its index.
- The index begins with 0 and ends at (total array size)-1.
- All the elements of the array can be accessed using Java for Loop.



Example: `// accessing the elements of the specified array`

```
for (int i = 0; i<arr.length; i++)
```

```
System.out.println("Element at index " + i + " : "+ arr[i]);
```

Points to Remember:

Unit 02: Arrays and Strings

- The elements in the array allocated by *new* will automatically be initialized to **zero** (for numeric types), **false** (for boolean), or **null** (for reference types).
- Obtaining an array is a two-step process. First, you must declare a variable of the desired array type. Second, you must allocate the memory that will hold the array, using *new*, and assign it to the array variable. Thus, in Java, all arrays are dynamically allocated.

Array Literal

In a situation, where the size of the array and variables of array are already known, array literals can be used.



Example: `int[] intArray = new int[]{ 1,2,3,4,5,6,7,8,9,10 }; // Declaring array literal`



Example:

```
// Program implementing 1dimensional Array
class arrayExample{
public static void main(String args[]){
int a[]=new int[5]; // declaration and instantiation
a[0]=10; // initialization
a[1]=20;
a[2]=30;
a[3]=40;
a[4]=50;
//traversing array
for(int i=0;i<a.length;i++) //length is the property of array
System.out.println(a[i]);
}}
```

Output

```
10
20
30
40
50
```



Task: Write a program to find the maximum and minimum values in an array.

2.3 Two Dimensional Array

In Java, multidimensional arrays are also referred to as arrays of arrays. A 2D array is a commonly used and simplest multi-dimensional array. Generally, 2D arrays are referred to as one-dimensional arrays' lists. 2D arrays are represented in a row-column form on paper, and the terms "rows" and "columns" are used in computing.

Syntax to Declare 2D Array

```
type array_name = new type[rows][cols];
```

In this syntax, *type* specifies the type (data type) of an array, the *array_name* specifies the name of the array, *rows*, and *cols* specify the number of rows and columns in the array. In a 2D array, memory needs to be allocated for the first dimension only and the remaining dimensions can be allocated separately.

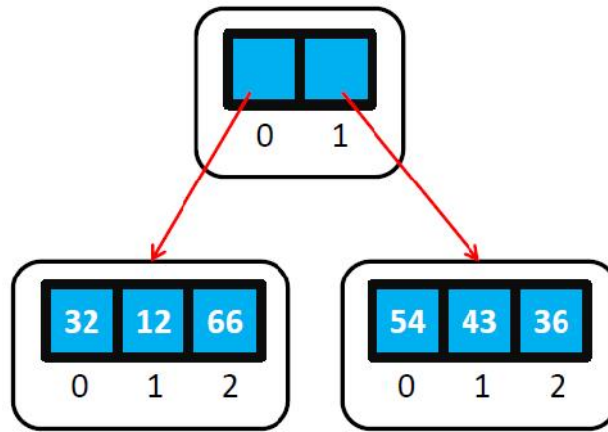


Figure 4: 2-Dimensional Array

```
int[ ][ ] har=new int[2][3];
```

```
har[0][0]=32;
```

```
har[0][1]=12;
```

```
har[0][2]=66;
```

```
har[1][0]=54;
```

```
har[1][1]=43;
```

```
har[1][2]=36;
```

Two-dimensional arrays can be created in two ways:

1. Reserving a block of memory that has enough space for holding all the array elements.
2. Building a multi-dimensional array from many one-dimensional arrays.

Example: `int two-dimensional [ ][ ] = new int [2] [3];`

This statement allocates a 2 by 3 dimensional array and assigns it to two-dimensional. This 2D array is implemented as an intarray of intarrays.



Example

```
//Java Program to implement 2-dimensional array
```

```
class Testarray3{
```

```
public static void main(String args[]){
```

```
//declaring and initializing 2D array
```

```
int arr[][]={{1,2,3},{4,5,6},{7,8,9}};
```

```
//printing 2D array
```

```
for(int i=0;i<3;i++){
```

```
for(int j=0;j<3;j++){
```

```
System.out.print(arr[i][j]+" ");
```

```
}
```

```
System.out.println();
```

```
}
```

```
}}
```

Output:

```
1 2 3
```

4 5 6

7 8 9

Multidimensional Arrays can be defined in simple words as an array of arrays. Data in multidimensional arrays are stored in tabular form (in row-major order).

Syntax:

```
data_type[1st dimension][2nd dimension][...][Nth dimension] array_name = new
data_type[size1][size2]...[sizeN];
```

where:

- **data_type**: Type of data to be stored in the array. For example: int, char, etc.
- **dimension**: The dimension of the array created. For example: 1D, 2D, etc.
- **array_name**: Name of the array
- **size1, size2, ..., sizeN**: Sizes of the dimensions respectively.



Task: Write a program to multiply two matrices and print the product.

Size of multidimensional arrays

The total number of elements that can be stored in a multidimensional array can be calculated by multiplying the size of all the dimensions.



Example: The array `int[ ][ ] x = new int[10][20]` can store a total of $(10 \times 20) = 200$ elements.

2.4 Strings

A string is a sequence of characters but it's not a primitive type. When we create a string in java, it creates an object of type String. A string is an immutable object which means that it cannot be changed once it is created. A string is the only class where operator overloading is supported in java. We can concat two strings using the + operator. For example "a"+"b"="ab". Java provides two useful classes for String manipulation - StringBuffer and StringBuilder.

Strings in Java are Objects that are backed internally by a char array. Since arrays are immutable (cannot grow), Strings are immutable as well. Whenever a change to a String is made, an entirely new String is created. A String variable contains a collection of characters surrounded by double-quotes.

Syntax:

```
<String_Type><string_variable> = "<sequence_of_string>";
```



Example: `String str = "Welcome";`

Memory Allocation to Strings

JVM divides the allocated memory to a Java program into two parts.

- **Stack**
- **heap.**

Stack is used for execution purposes and heap is used for storage purposes. In that heap memory, JVM allocates some memory specially meant for string literals. This part of the heap memory is called String Constant Pool. Whenever you create a string object using a string literal, that object is stored in the string constant pool. Whenever you create a string object using a new keyword, such object is stored in the heap memory.



Example

- `String s1 = "abc";`
- `String s2 = "xyz";`

Programming in JAVA

- `String s3 = "123";`
- `String s4 = "A";`

will be stored in the String Constant Pool.

When you create string objects using new keyword like below, they will be stored in the heap memory.



Example:

- `String s5 = new String("abc");`
- `char[] c = {'J', 'A', 'V', 'A'};`
- `String s6 = new String(c);`
- `String s7 = new String(new StringBuffer());`

One more interesting thing about String Constant Pool is that **pool space is allocated to an object depending upon its content**. There will be no two objects in the pool having the same content.

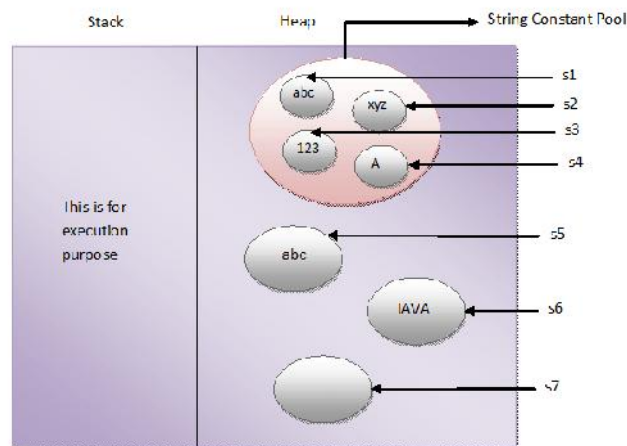


Figure 5: Data in String Constant Pool

2.5 Different Ways of Creating Strings

The following are the ways to create a string object:

1. Using string literal.
2. Using a new keyword.

Using string literal

This is the most common way of creating a string. In this case, a string literal is enclosed with double-quotes.

Syntax:

`String var_name="value"`



Example:

`String str = "abc";`

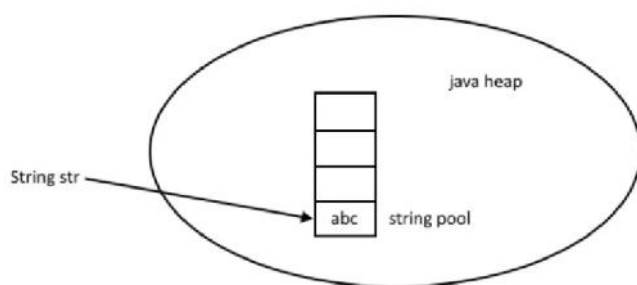


Figure 6: Internal Working

When we create a String using double quotes, JVM looks in the String pool to find if any other String is stored with the same value. If found, it just returns the reference to that String object else it creates a new String object with a given value and stores it in the String pool.

Using new keyword

We can create String object using new operator, just like any normal java class. There are several constructors available in String class to get String from char array, byte array, StringBuffer and StringBuilder.



Example:

```
String str = new String("abc");
```

```
char[ ] a = {'a', 'b', 'c'};
```

```
String str2 = new String(a);
```

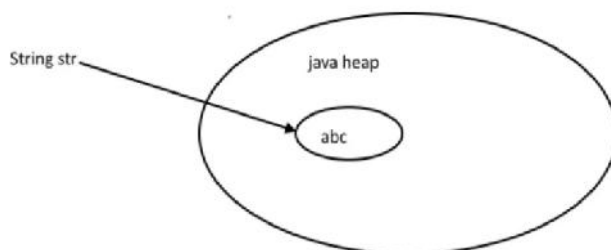


Figure 7: Internal Working Using new



Example

```
public class StringDemo {
    public static void main(String args[ ]) {
        char[ ] helloArray = { 'h', 'e', 'l', 'l', 'o', ' ' };
        String helloString = new String(helloArray);
        System.out.println( helloString );
    }
}
```

Output

Hello

2.6 String Methods

A String in Java is an object, which contains methods that can perform certain operations on strings which are as follows:

- String Length.
- String Concatenation.
- Finding a Character in a String.
- To convert the case of a String.
- Comparison of Strings.
- contains

String Length

The accessor method that you can use with strings is the `length()` method, which returns the number of characters contained in the string object. Methods used to obtain information about an object are known as **accessor methods**.



Example:

```
public class Main {
    public static void main(String[] args) {
        String txt = "ABCDEFGHJKLMNOPQRSTUVWXYZ";
        System.out.println("The length of the txt string is: " + txt.length());
    }
}
```

Output: 26

String Concatenation

- Strings are more commonly concatenated with the `+` operator
- The `+` operator can be used between strings to combine them.



Example:

```
"Hello," + " world" + "!"
results in -"Hello, world!"

public class StringDemo {
    public static void main(String args[]) {
        String string1 = "saw I was ";
        System.out.println("Dot " + string1 + "Tod");
    }
}
```

Output:

Dot saw I was Tod

You can also use the `concat()` method with string literals for concatenation.

Syntax:

```
string1.concat(string2);
```

This returns a new string that is `string1` with `string2` added to it at the end.

**Example:**

```
"My name is ".concat("Zara");
```

Finding a Character in a String

The `indexOf()` method returns the index (the position) of the first occurrence of a specified text in a string (including whitespace).

**Example:**

```
public class Main {
    public static void main(String[] args) {
        String txt = "Please locate where 'locate' occurs!";
        System.out.println(txt.indexOf("locate"));
    }
}
```

Output: 7 //Java counts positions from zero. 0 is the first position in a string, 1 is the second, 2 is the third .

To convert the case of a String

`toUpperCase()` and `toLowerCase()` are used to convert the case of a string.

**Example:**

```
String txt = "Hello World";
System.out.println(txt.toUpperCase()); // Outputs "HELLO WORLD"
System.out.println(txt.toLowerCase()); // Outputs "hello world"
```

Comparison of Strings

String class provides `equals()` and `equalsIgnoreCase()` methods to compare two strings. These methods compare the value of a string to check if two strings are equal or not. It returns true if two strings are equal and false if not.

**Example**

```
public class StringEqualExample
{
    public static void main(String[] args)
    { //creating two string object
        String s1 = "abc";
        String s2 = "abc";
        String s3 = "def";
        String s4 = "ABC";
        System.out.println(s1.equals(s2)); // true
        System.out.println(s2.equals(s3)); // false
        System.out.println(s1.equals(s4)); // false;
        System.out.println(s1.equalsIgnoreCase(s4)); // true } }
```

String class implements Comparable interface, which provides `compareTo()` and `compareToIgnoreCase()` methods and it compares two strings lexicographically. Both strings are converted into Unicode values for comparison and return an integer value that can be greater than,

Programming in JAVA

less than, or equal to zero. If strings are equal then it returns zero or else it returns either greater or less than zero.



Example

```
public class StringCompareToExample {
    public static void main(String[] args)
    {
        String a1 = "abc";
        String a2 = "abc";
        String a3 = "def";
        String a4 = "ABC";
        System.out.println(a1.compareTo(a2)); // 0
        System.out.println(a2.compareTo(a3)); // less than 0
        System.out.println(a1.compareTo(a4)); // greater than 0
        System.out.println(a1.compareToIgnoreCase(a4)); // 0
    }
}
```

Contains

Java String contains() methods that check if the string contains a specified sequence of characters or not. This method returns true if the string contains a specified sequence of characters, else returns false.



Example

```
public class StringContainsExample
{
    public static void main(String[] args)
    {
        String s = "Hello World"; System.out.println(s.contains("W")); // true
        System.out.println(s.contains("X")); // false
    }
}
```

2.7 String Array

As the name suggests, a string array is an array containing strings. We can declare string arrays in the following two ways:

1. With an initial size
2. Without an initial size

1. *With an Initial Size:*

In Java, we can declare string arrays and assign an initial size to them.

```
public StringArrayDemo
{
    private String[ ] habit = new String[10];
    // more to the class here ...
}

void populateStringArray()
```

```
{
habit[0] = "Hello";
habit[1] = " Welcome";
habit[2] = "Great";
// ...
}
```

In this example,

1. A class StringArrayDemo is created and declared public using the public keyword.
2. In the class JavaStringArrayDemo, a String array named as fruits is created, where the habitarray has been given an initial size of 10 elements.
3. Then, the elements in the String array are assigned by the populateStringArray() method in the class:
 - (a) habit[0] is assigned a string Hello.
 - (b) habit[1] is assigned a string Welcome.
 - (c) habit[2] is assigned a string Great.



Did you Know?

A String array in Java begins with an element numbered zero.

2. Without an Initial Size:

We can also declare a Java String array without giving it an initial size.

```
public class JavaStringArrayDemo
{
private String[ ] toppings;
// more to the class here ...
}
```

After this, the Java array can be given size in the program code, and populated as desired, like this:

```
void populateStringArray( )
{
fruits[0] = "Apple";
fruits[1] = "Mango";
fruits[2] = "Banana";
// ...
}
```

This method of declaring an array is very similar to the first method. However, in this method, the string array is not given any size until the populateStringArray method is called.

2.8 StringBuffer Class

Java StringBuffer class is used to create a mutable string. This class is the same as the String class except it is mutable i.e. it can be changed. StringBuffer may have characters and substrings inserted in the middle or appended to the end. Following are the important points about StringBuffer:

- A string buffer is like a String but can be modified.

Programming in JAVA

- It contains some particular sequence of characters, but the length and content of the sequence can be changed through certain method calls.
- They are safe for use by multiple threads.
- Every string buffer has a capacity.

Table 1: StringBuffer Constructors

Constructor	Description
StringBuffer()	creates an empty string buffer with the initial capacity of 16.
StringBuffer(String str)	creates a string buffer with the specified string.
StringBuffer(int capacity)	creates an empty string buffer with the specified capacity as length.

StringBuffer() : It reserves room for 16 characters without reallocation.



Example:StringBuffer s=new StringBuffer();

StringBuffer(int size): It accepts an integer argument that explicitly sets the size of the buffer.



Example:StringBuffer s=new StringBuffer(20);

StringBuffer(String str): It accepts a String argument that sets the initial contents of the StringBuffer object and reserves room for 16 more characters without reallocation.



Example:StringBuffer s=new StringBuffer("Welcome");

The following table shows the various methods used in the StringBuffer class:

Table 2: Methods of StringBuffer class

Method	Description
append(String s)	It is used to append the specified string with this string.
insert(int offset, String s)	It is used to insert the specified string with this string at the specified position.
replace(int startIndex, int endIndex, String str)	is used to replace the string from specified startIndex and endIndex.
delete(int startIndex, int endIndex)	is used to delete the string from specified startIndex and endIndex.
reverse()	It is used to reverse the string.
capacity()	It is used to return the current capacity.
ensureCapacity(int minimumCapacity)	It is used to ensure the capacity is at least equal to the given minimum.
charAt(int index)	It is used to return the character to the specified position.
length()	It is used to return the length of the string i.e.

Unit 02: Arrays and Strings

	total number of characters.
substring(int beginIndex)	It is used to return the substring from the specified beginIndex.
substring(int beginIndex, int endIndex)	It is used to return the substring from the specified beginIndex and endIndex.



Example

//Program to implement StringBufferappend() method

```

class StringBufferExample{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello ");
sb.append("Java");//now original string is changed
System.out.println(sb);//prints Hello Java
}
}

```



Example

// Program to implement StringBufferinsert() method

```

class StringBufferExample2{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello ");
sb.insert(1,"Java");//now original string is changed
System.out.println(sb);//prints HJavaello
}
}

```



Example

// Program to implement StringBufferreplace() method

```

class StringBufferExample3{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.replace(1,3,"Java");
System.out.println(sb); //prints HJavalo
}
}

```



Example

// Program to implement StringBufferdelete() method

```

class StringBufferExample4{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.delete(1,3);
}
}

```

Programming in JAVA

```
System.out.println(sb); // prints Hlo
}
}
```



Example

```
// Program to implement StringBufferreverse() method
```

```
class StringBufferExample5{
    public static void main(String args[]){
        StringBuffer sb=new StringBuffer("Hello");
        sb.reverse();
        System.out.println(sb); // prints olleH
    }
}
```



Example

```
// Program to implement StringBuffercapacity() method
```

```
class StringBufferExample6{
    public static void main(String args[]){
        StringBuffer sb=new StringBuffer();
        System.out.println(sb.capacity()); // default 16
        sb.append("Hello");
        System.out.println(sb.capacity()); // now 16
        sb.append("java is my favourite language");
        System.out.println(sb.capacity()); // now (16*2)+2=34 i.e (oldcapacity*2)+2
    }
}
```



Example

```
// Program to implement StringBufferensureCapacity() method
```

```
class StringBufferExample7{
    public static void main(String args[]){
        StringBuffer sb=new StringBuffer();
        System.out.println(sb.capacity()); // default 16
        sb.append("Hello");
        System.out.println(sb.capacity()); // now 16
        sb.append("java is my favourite language");
        System.out.println(sb.capacity()); // now (16*2)+2=34 i.e (oldcapacity*2)+2
        sb.ensureCapacity(10); // now no change
        System.out.println(sb.capacity()); // now 34
        sb.ensureCapacity(50); // now (34*2)+2
        System.out.println(sb.capacity()); // now 70
    }
}
```

}

2.9 StringBuilder Class

Java `StringBuilder` class is used to create a mutable (modifiable) string. The Java `StringBuilder` class is the same as the `StringBuffer` class except that it is non-synchronized. It is available since JDK 1.5. `StringBuilder` class provides an API similar to `StringBuffer`, but unlike `StringBuffer`, it doesn't guarantee thread safety. The following table shows the various constructors of the `StringBuilder` class:

Table 3: *StringBuilder Constructors*

Constructor	Description
<code>StringBuilder()</code>	Creates an empty string builder with a default capacity of 16 (16 empty elements).
<code>StringBuilder(CharSequence cs)</code>	Constructs a string builder containing the same characters as the specified <code>CharSequence</code> , plus an extra 16 empty elements trailing the <code>CharSequence</code> .
<code>StringBuilder(int initCapacity)</code>	Creates an empty string builder with the specified initial capacity.
<code>StringBuilder(String s)</code>	Creates a string builder whose value is initialized by the specified string, plus an extra 16 empty elements trailing the string.

StringBuilder Methods

The following are the various methods used by `StringBuilder` class:

StringBuilder Length and Capacity

```
// creates empty builder, capacity 16
StringBuilder sb = new StringBuilder();
// adds 5 character string at beginning sb.append("Hello");
System.out.println("StringBuilder length = "+sb.length()); // prints 5
System.out.println("StringBuilder capacity = "+sb.capacity()); // prints 16
```

Append()

```
public class StringBuilderExample
{
    public static void main(String[] args)
    {
        StringBuilder sb = new StringBuilder("Hello "); sb.append("World");// now original string is changed
        System.out.println(sb); // prints Hello World
    }
}
```

Insert()

```
StringBuilder sb = new StringBuilder("HellWorld");
```

Programming in JAVA

```
sb.insert(4, "o ");
```

```
System.out.println(sb); // prints Hello World
```

replace(int startIndex, int endIndex, String str)

```
StringBuilder sb = new StringBuilder("Hello World!");
```

```
sb.replace(6,11,"Earth");
```

```
System.out.println(sb); // prints Hello Earth!
```

delete(int startIndex, int endIndex)

```
StringBuilder sb = new StringBuilder("Journalgood.com");
```

```
sb.delete(7,14);
```

```
System.out.println(sb); // prints Journal
```

Capacity()

```
StringBuilder sb=new StringBuilder();
```

```
System.out.println(sb.capacity()); // default value 16
```

```
sb.append("Java");
```

```
System.out.println(sb.capacity()); // still 16
```

```
sb.append("Hello StringBuilder Class!");
```

```
System.out.println(sb.capacity()); // (16*2)+2
```

Reverse()

```
StringBuilder sb = new StringBuilder("lived");
```

```
sb.reverse();
```

```
System.out.println(sb); // prints devil
```

Table 4: StringBuffer v/s StringBuilder

StringBuffer	StringBuilder
StringBuffer is <i>synchronized</i> i.e. thread-safe. It means two threads can't call the methods of StringBuffer simultaneously.	StringBuilder is <i>non-synchronized</i> i.e., not thread-safe. It means two threads can call the methods of StringBuilder simultaneously.
Operates slower due to thread safety feature	Better performance compared to StringBuffer
Has some extra methods - substring, length, capacity, etc.	Not needed because these methods are present in String too.
Introduced in Java 1.2	Introduced in Java 1.5 for better performance.
StringBuffer is <i>less efficient</i> than StringBuilder.	StringBuilder is <i>more efficient</i> than StringBuffer.

2.10 Access Specifiers

The access modifiers in Java specify the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and classes by applying the access modifier to them. There are two types of modifiers in Java:

- ✓ **access specifiers:** controls the access level.
- ✓ **non-access specifiers:** do not control access level, but provide other functionality

For classes, you can use either public or default. For attributes, methods, and constructors, you can use one of the following:

- ✓ **public** -The code is accessible for all classes
- ✓ **private** -The code is only accessible within the declared class
- ✓ **default** -The code is only accessible in the same package. This is used when you don't specify a modifier.
- ✓ **protected** -The code is accessible in the same package and **subclasses**.

Table 5: Access Specifiers

Access Modifier	Within class	Within package	Outside package by subclass only	Outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y



Example

//Program for the implementation of Private Access Specifier

```

class A{
    private int data=40;
    private void msg()
    {
        System.out.println("Hello java");
    }
}

public class Simple{
    public static void main(String args[])
    {
        A obj=new A();
        System.out.println(obj.data);//Compile Time Error
        obj.msg();//Compile Time Error
    }
}

```

In this example, we have created two classes A and Simple. A class contains private data members and private methods. We are accessing these private members from outside the class, so there is a compile-time error.



Example

//Program for the implementation Default Access Specifier

```

package pack;

class A{
    void msg(){System.out.println("Hello");
}

```

```
}
```

In the example, the scope of class A and its method msg() is the default so it cannot be accessed from outside the package.

```
package mypack;
import pack.*;
class B{
    public static void main(String args[]){
        A obj = new A();
        //Compile Time Error
        obj.msg();
        //Compile Time Error
    }
}
```

In this example, we have created two packages pack and a mypack. We are accessing the A class from outside its package, since the A class is not public, so it cannot be accessed from outside the package.



Example

//Program for the implementation protected Access Specifier

```
package pack;
public class A{
    protected void msg()
    {
        System.out.println("Hello");} }
package mypack;
import pack.*;
class B extends A{
    public static void main(String args[]){
        B obj = new B();
        obj.msg();
    }
}
```

Output: hello

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.



Example

//Program for the implementation of public Access Specifier

```
package pack;
public class A{
    public void msg()
```

```

{
    System.out.println("Hello");
}

package mypack;
import pack.*;

class B{
    public static void main(String args[])
    {
        A obj = new A();
        obj.msg();
    }
}

```

Output: Hello

The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

2.11 Inheritance

Inheritance is an important pillar of OOP(Object-Oriented Programming). It is the mechanism in java by which one class is allowed to inherit the features(fields and methods) of another class. The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields to your current class also. Inheritance represents the IS-A relationship which is also known as a *parent-child* relationship.

Syntax:

```

class Subclass-name extends Superclass-name
{
    //methods and fields
}

```

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality

Important Terms

Class: A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.

Sub Class/Child Class: A subclass is a class that inherits the other class. It is also called a derived class, extended class, or child class.

Super Class/Parent Class: Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.

Reusability: As the name specifies, reusability is a mechanism that facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

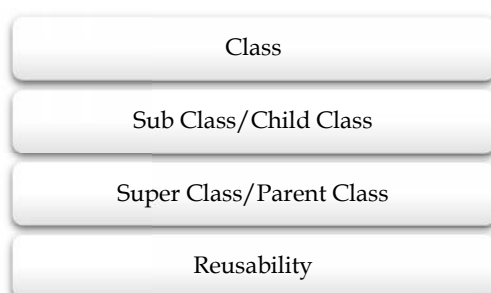


Figure 8: Terms of Inheritance

Types of inheritance

The following figure shows the various types of inheritance available in java.

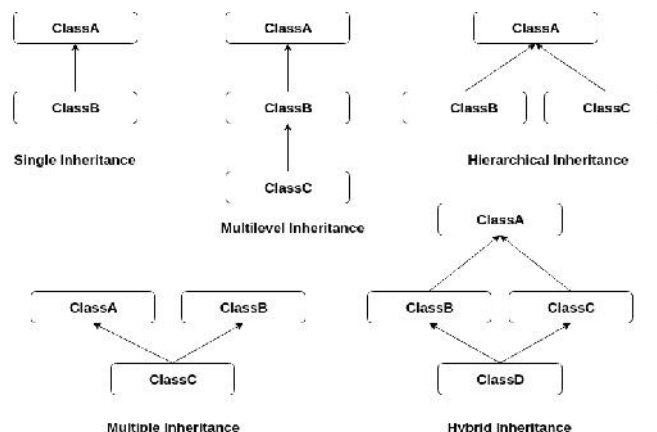


Figure 9: Inheritance Types

Inheritance is one of the object-oriented concepts. It is a process, where one object inherits the properties of another. Similarly, class inheritance means that a class derives a set of properties and methods of a parent class or base class. To inherit a base class to its subclass, a keyword `extends` is used in the subclass definition.

Single Inheritance

When a subclass is derived directly from its parent class or superclass, it is known as simple inheritance. In simple inheritance, there is only a subclass and its superclass. It is also referred to as single inheritance or one-level inheritance.

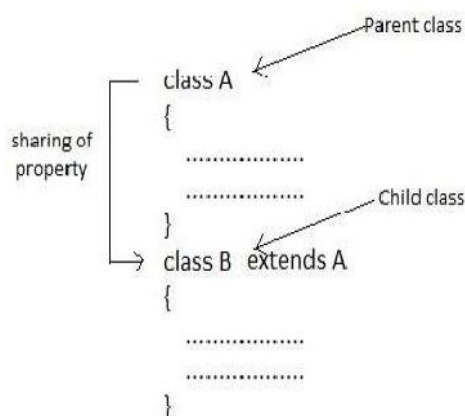


Figure 10: Single Inheritance



Example Program to illustrate the use of simple inheritance in a program.

```

public class Superclass
{
    public void A()
    {
        System.out.println("Print the super class method");
    }
}

class Subclass extends Superclass
{
    public static void main(String args[] )
    {
        Subclass s1 = new Subclass();
        s1.A();
    }
}
  
```

Output:

Print the super class method



Task: Write a simple inheritance program to show one subclass manager and its superclass employee.

Multilevel Inheritance

Multi-level inheritance was introduced to enhance the concept of inheritance. When a subclass is derived from another subclass or derived class, it is known as the multi-level inheritance. In multi-level inheritance, the subclass is the child class for its superclass and this superclass is the subclass for another superclass. Multi-level inheritance can go up to any number of levels.

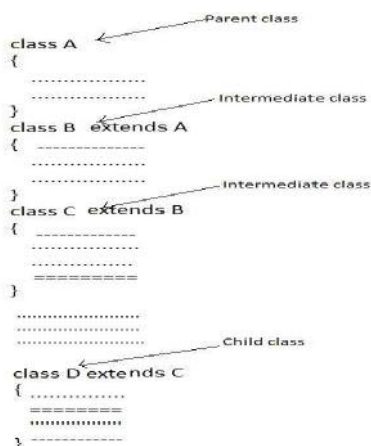


Figure 11: Multi-Level Inheritance



Example

```

class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark()
{System.out.println("barking...");}
}
class BabyDog extends Dog{
void weep()
{System.out.println("weeping...");}
}
class TestInheritance2{
public static void main(String args[])
{
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}}
  
```

Programming in JAVA

As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

weeping...

barking...

eating...

Hierarchal Inheritance

In Hierarchical Inheritance, the multiple child classes inherit the single class, or the single class is inherited by multiple child classes. A structure having one parent and more child class. Child classes must be connected with only Parent classes.

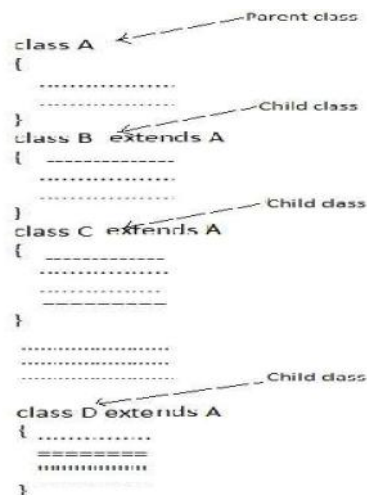


Figure 12: Hierarchical Inheritance

Multiple Inheritance

When the child class extends from more than one superclass, it is known as multiple inheritance. However, Java does not support multiple inheritance. To achieve multiple inheritance in Java, we must use the interface.

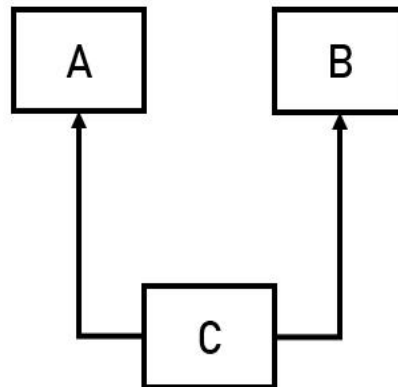


Figure 13: Multiple Inheritance



Example

```

interface AnimalEat {
    void eat();
}

interface AnimalTravel {
    void travel();
}
  
```

```

}
class Animal implements AnimalEat, AnimalTravel
{
    public void eat() {
        System.out.println("Animal is eating");
    } public void travel() {
        System.out.println("Animal is travelling");
    }
}
public class Demo {
    public static void main(String args[])
    {
        Animal a = new Animal();
        a.eat();
        a.travel();
    }
}

```

The interface `AnimalEat` and `AnimalTravel` have one abstract method each i.e. `eat()` and `travel()`. The class `Animal` implements the interfaces `AnimalEat` and `AnimalTravel`. In the method `main()` in class `Demo`, an object of class `Animal` is created. Then the methods `eat()` and `travel()` are called.

Hybrid Inheritance

Hybrid Inheritance is a combination of both **Single Inheritance** and **Multiple Inheritance**. Since java doesn't support multiple inheritances with classes, hybrid inheritance is also not possible with classes. In java, we can achieve hybrid inheritance only through Interfaces.

Important facts about inheritance

Default superclass: Except `Object` class, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of the `Object` class.

Superclass can only be one: A superclass can have any number of subclasses. But a subclass can have only **one** superclass. This is because Java does not support multiple inheritances with classes.

Inheriting Constructors: A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

Private member inheritance: A subclass does not inherit the private members of its parent class. However, if the superclass has public or protected methods (like getters and setters) for accessing its private fields, these can also be used by the subclass.

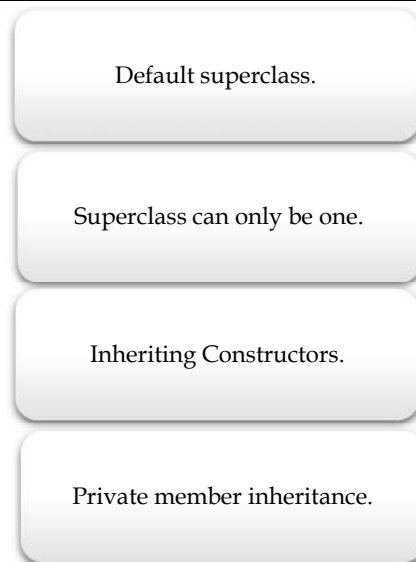


Figure 14: Inheritance Facts

**Lab Exercise**

1. Write a program to print the sum of two matrices.
2. Write a program to swap the values of two arrays, and print the output.

Summary

- In a Java program, the length of an array is fixed and it starts with zero (0).
- An array can be created with or without using the new operator.
- Initialization of an array can be done at the time of array declaration, or after the array creation.
- Array length specifies the number of array elements in an array.
- One-dimensional arrays are defined as a variables' list comprising the data of the same type.
- Generally, two-dimensional or 2D arrays are referred to as one-dimensional arrays' lists. 2D arrays are represented in a row-column form
- A string is a sequence of characters that is created using the String class.
- The length of a string refers to the number of characters in a string.
- An array that contains strings is a string array.
- The string class consists of some methods for creating string objects. These methods are called String methods.
- The StringBuffer class is a peer class of String, which is used for strings' alteration.
- To inherit a base class to its subclass, a keyword extends is used in the subclass definition.
- We can change the access level of fields, constructors, methods, and classes by applying the access modifier to them

Keywords

StringBuilder(): Creates an empty string builder with a default capacity of 16 (16 empty elements).

Access Specifier: Controls the access to the names that follow it, up to the next access specifier or the end of the class declaration.

Unit 02: Arrays and Strings

Hierarchical Inheritance: Multiple classes being derived from a single class.

Class: A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.

Array: Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

Hybrid Inheritance: is a combination of both Single Inheritance and Multiple Inheritance.

concat(): The String concat() method combines a specific string at the end of another string and ultimately returns a combined string.

StringBuilder(): Creates an empty string builder with a default capacity of 16 (16 empty elements).

Super Class/Parent Class: Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.

Reusability: As the name specifies, reusability is a mechanism that facilitates you to reuse the fields and methods of the existing class when you create a new class.

Self Assessment

1. Which of the following methods is used to create object a's string representation?

- A. a.toString()
- B. StringTokenizer.countTokens()
- C. str1.append(str2)
- D. str2 = str1.replace('a', 'b');

2. Which of these methods of String class can be used to test strings for equality?

- A. isequal()
- B. isequals()
- C. equal()
- D. equals()

3. What will be the output of the following Java program?

```
class string_demo
{
    public static void main(String args[])
    {
        String obj = "I" + "like" + "Java";
        System.out.println(obj);
    }
}
```

- A. I
- B. like
- C. Java
- D. IlikeJava

4. What will be the output of the following Java program?

```
class string_class
{
```

Programming in JAVA

```
public static void main(String args[])  
{  
    String obj = "I LIKE JAVA";  
    System.out.println(obj.charAt(3));  
}  
}
```

- A. I
- B. L
- C. K
- D. E

5. Which of these operators can be used to concatenate two or more String objects?

- A. +
- B. +=
- C. &
- D. ||

6. Which of this methods of class String is used to obtain a length of String object?

- A. get()
- B. Sizeof()
- C. lengthof()
- D. length()

7. Which of these classes is a superclass of String and StringBuffer class?

- A. java.util
- B. java.lang
- C. ArrayList
- D. None of the mentioned

8. What is the value returned by unction compareTo() if the invoking string is less than the string compared?

- A. zero
- B. a value less than zero
- C. a value greater than zero
- D. None of the mentioned

9. Which of these data type value is returned by the equals() method of String class?

- A. char
- B. int
- C. boolean
- D. All of the mentioned

10. In Java arrays are

- A. objects
- B. object references
- C. primitive data type
- D. None of the above

11. Which one of the following is a valid statement?

- A. `char[] c = new char();`
- B. `char[] c = new char[5];`
- C. `char[] c = new char(4);`
- D. `char[] c = new char[];`

12. Arrays in Java are dynamically allocated using the operator.

- A. create
- B. dynamic
- C. ArrayList
- D. New

13. To declare a one-dimensional array, you will use this general form

- A. `type array-name[] = new [size];`
- B. `type array-name[size] = new type[];`
- C. `type array-name[] = new type[size];`
- D. `type array-name[] = type[size];`

14. In Java, each array object has a final field named that stores the size of the array.

- A. width
- B. size
- C. length
- D. distance

15. In Java language, an array index starts with ____.

- A. -1
- B. 0
- C. 1
- D. Any integer

Answers for Self Assessment

- | | | | | |
|-------|-------|-------|-------|-------|
| 1. D | 2. D | 3. D | 4. A | 5. A |
| 6. D | 7. B | 8. B | 9. C | 10. A |
| 11. B | 12. D | 13. C | 14. C | 15. B |

Review Questions

1. "In an array, the memory is allocated for the same data type sequentially and is given a common name." Justify.
2. "An array can also be created without using the new operator as Java supports dynamic array allocation." Justify with an example.
3. "After the creation of an array, it should be initialized and given a value." Discuss.
4. "Various string methods are used for different tasks of string manipulation." Discuss those methods with examples.
5. "The StringBuffer class consists of some methods that can be used for the manipulation of the objects of that class." Elaborate.
6. With an example explain the various access specifiers.
7. Why do we need to use inheritance? Explain the concept of superclass and subclass?

**Further Readings**

Balagurusamy E. Programming with Java 3e Primer. New Delhi: Tata McGrawPublishers.

Schildt H. Java A Beginner's Guide, 3rd ed. New York: Mc-Graw Hill.

Arnold, K., Gosling, J., & Holmes, D. (2005). *The Java programming language*. Addison WesleyProfessional.

Haggar, P. (2000). *Practical Java: programming language guide*. Addison-Wesley Professional.

**Web Links**

<http://www.javabeginner.com/learn-java/java-string-comparison>

<http://www.leepoint.net/notesjava/data/strings/55stringTokenizer/10stringtokenizer.html>

http://admashmc.com/main/images/Lec_Notes/javaarray.pdf

<https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html>

https://www.w3schools.com/java/java_arrays.asp

<https://www.baeldung.com/java-arrays-guide>

Unit 03: Collection Framework

CONTENTS

Objectives

Introduction

3.1 ArrayList Class

3.2 ArrayList Constructors

3.3 ArrayList Operations

3.4 ListIterator

3.5 LinkedList Class

3.6 TreeSet Class

3.7 PriorityQueue class

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

Objectives

- Learn the basic concept of ArrayList Class, listiterator
- Understand the different constructors and basic operations of ArrayList Class
- Analyze the different methods of the listiterator.
- Learn the basic concept of LinkedList class.
- Understand the different constructors of the LinkedList class.
- Analyze the basic methods of the LinkedList class.
- Know the concept of TreeSet class, its constructors, and methods.
- Explore the concept of PriorityQueueclass and its various methods

Introduction

An ArrayList in Java represents a resizable list of objects. We can add, remove, find, sort, and replace elements in this list. It is part of the collections framework. It extends AbstractList which implements the List interface. The List extends Collection and Iterable interfaces in hierarchical order. ArrayList's give great performance on get() and set() methods, but do not perform well on add() and remove() methods when compared to a LinkedList.

3.1 ArrayList Class

ArrayList class uses a *dynamic array* for storing the elements. It is like an array, but there is *no size limit*. We can add or remove elements anytime. So, it is much more flexible than the traditional array. It is found in *java.util* package.



Implementation cycle of ArrayList

ArrayList is a part of the collectionframework and is present in java.util.package. The ArrayList class extends AbstractList and implements the List interface. An ArrayLists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array may be shrunk.

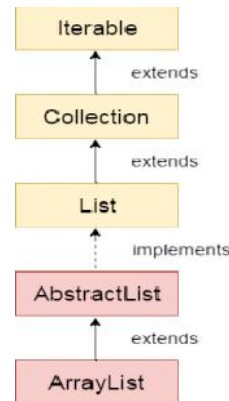


Figure 1: Hierarchy of ArrayList Class

Features of ArrayList

- ArrayList inherits AbstractList class and implements the List interface.
- ArrayList is initialized by the size. Elements can be randomly accessed using index positions. Index starts with '0'.
- In ArrayList, the size is increased automatically if the collection grows or shrinks if the objects are removed from the collection.
- ArrayList can not be used for primitive types, like int, char, etc. We need a wrapper class for such cases.
- ArrayList in Java can be seen as a vector in C++.
- ArrayList is not Synchronized. Its equivalent synchronized class in Java is Vector.

Creation of ArrayList

```
import java.util.ArrayList;
```

```
ArrayList<String> cars = new ArrayList<String>();
```

Points to Remember

- ArrayList class can contain duplicate elements.
- ArrayList class maintains insertion order.
- ArrayList class is non-synchronized.
- ArrayList allows random access because the array works on an index basis.
- In ArrayList, manipulation is a little bit slower because a lot of shifting needs to occur if any element is removed from the array list.

3.2 ArrayList Constructors

To create ArrayList, we can call one of its constructors. The following table shows the various types of constructors along with their descriptions.

Table 1: ArrayList Constructors

Constructor	Description
ArrayList()	This constructor builds an

	empty array list.
ArrayList(Collection c)	This constructor builds an array list that is initialized with the elements of the collection c .
ArrayList(int capacity)	This constructor builds an array list that has the specified initial capacity.

3.3 ArrayList Operations

The following are the basic operations that we can perform using ArrayList.

1. **Adding Elements:** Here we are discussing about add() method of Java.util.ArrayList class.

This method is used for adding an element to the ArrayList.

- add(Object) : This method adds an object to the arraylist.
- add(int index, Object) : It adds the object to the array list at the given index.



Example

// Program to add elements in an ArrayList

```
import java.util.ArrayList;

public class ArrList {

    public static void main(String[] args) {

        ArrayList<String> cars = new ArrayList<String>();

        cars.add("Maruti");
        cars.add("Ford");
        cars.add("Mahindra");
        cars.add("Mazda");

        System.out.println(cars);

    }
}
```

Output

```
PS C:\Users\ OneDrive\Documents\cap615>javac ArrList.java
```

```
PS C:\Users\ OneDrive\Documents\cap615> java ArrList
```

```
[Maruti, Ford, Mahindra, Mazda]
```

2. **Changing Elements :** We can use the setmethod to change an element in ArrayList. We provide the index and new element, this method then updates the element present at the given index with the new given element.

```
import java.util.ArrayList;

public class Main {

    public static void main(String[] args) {

        ArrayList<String> cars = new ArrayList<String>();

        cars.add("Volvo");
        cars.add("BMW");
```

```
cars.add("Ford");
cars.add("Mazda");
cars.set(0, "Opel");
System.out.println(cars);
} }
```

Output:

```
PS C:\Users\ OneDrive\Documents\cap615>javac ChList.java
```

```
PS C:\Users\ OneDrive\Documents\cap615> java ChList
```

```
[Opel, BMW, Ford, Mazda]
```

3. Removing Elements :We use **remove()** method to remove elements from an ArrayList, Same as add() method, this method also has few variations.

- remove(Object): This method removes an object from the arraylist.
- remove(int index): It removes the object from the array list at the given index.



Example

```
//Program to implement remove() method
import java.util.ArrayList;
public class ReList {
    public static void main(String[] args) {
        ArrayList<String> cars = new ArrayList<String>();
        cars.add("Maruti");
        cars.add("BMW");
        cars.add("Ford");
        cars.add("Mazda");
        System.out.println("Before Removal\n" + cars);
        cars.remove(0);
        System.out.println("After Removal\n" + cars);
    } }
```

Output :

```
PS C:\Users\OneDrive\Documents\cap615>javac ReList.java
```

```
PS C:\Users\OneDrive\Documents\cap615> java ReList
```

```
Before Removal
```

```
[Maruti, BMW, Ford, Mazda]
```

```
After Removal
```

```
[BMW, Ford, Mazda]
```

4. Access an Item: ArrayListget(int index) method is used for fetching an element from the list. We need to specify the index while calling the get method and it returns the value present at the specified index. In the below example we are getting a few elements of an ArrayList by using the get method.



Example

```
// Implementation of get() method
```

```
import java.util.ArrayList;
public class AccList {
    public static void main(String[] args) {
        ArrayList<String> items = new ArrayList<String>();
        items.add("PEN");
        items.add("ERASER");
        items.add("FEVICOL");
        items.add("PENCIL");
        System.out.println(items);
        System.out.println(items.get(0));
    } }

```

Output :

```
PS C:\Users\ OneDrive\Documents\cap615>javac AccList.java
PS C:\Users\ OneDrive\Documents\cap615> java AccList
[PEN, ERASER, FEVICOL, PENCIL]
PEN

```

5. ArrayList Size: By using the size() method of the ArrayList class we can easily determine the size of the ArrayList. This method returns the number of elements of ArrayList.



Example:

```
//Implementation of size() method
import java.util.ArrayList;
public class SizeList {
    public static void main(String[] args) {
        ArrayList<String> items = new ArrayList<String>();
        items.add("PEN");
        items.add("ERASER");
        items.add("FEVICOL");
        items.add("PENCIL");
        System.out.println(items);
        System.out.println(items.get(0));
    } }

```

Output

```
PS C:\Users\ OneDrive\Documents\cap615>javac SizeList.java
PS C:\Users\ OneDrive\Documents\cap615> java SizeList
[PEN, ERASER, FEVICOL, PENCIL]
4

```

6. Loop Through an ArrayList:
7. Sort an ArrayList: An ArrayList can be sorted by using the sort() method of the Collections class in Java. It accepts an object of ArrayList as a parameter to be sorted and returns an ArrayList sorted in the ascending order according to the natural ordering of its elements.



Example

```
//Implementation of sort() method
import java.util.*;
import java.util.ArrayList;

```

```
public class SizeList {
    public static void main(String[] args) {
        ArrayList<String> items = new ArrayList<String>();
        items.add("PEN");
        items.add("ERASER");
        items.add("FEVICOL");
        items.add("PENCIL");
        System.out.println(items);
        System.out.println(items.size());
        System.out.println("Before Sorting: "+ items);
        // SortingArrayList in ascending Order
        Collections.sort(items);
        // printing the sorted ArrayList
        System.out.println("After Sorting: "+ items);
    } }

```

Output :

PS C:\Users\OneDrive\Documents\cap615>javac SizeList.java

PS C:\Users\ OneDrive\Documents\cap615> java SizeList

[PEN, ERASER, FEVICOL, PENCIL]

4

Before Sorting: [PEN, ERASER, FEVICOL, PENCIL]

After Sorting: [ERASER, FEVICOL, PEN, PENCIL]

3.4 ListIterator

The ListIterator interface of the Java collections framework provides the functionality to access elements of a list. It is bidirectional. This means it allows us to iterate elements of a list in both directions. It has no current element; its cursor position always lies between the element that would be returned by a call to the previous() and the element that would be returned by a call to next().

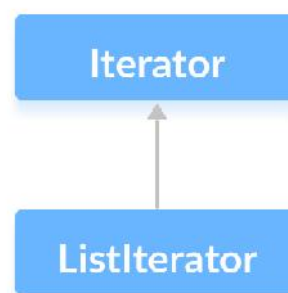
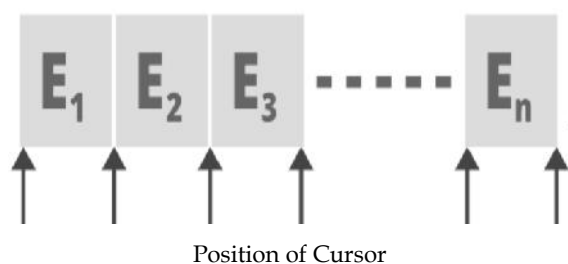


Figure 2: ListIterator extends the Iterator interface.

Features of ListIterator

- It is useful for listing implemented classes.
- Available since java 1.2.
- It supports bi-directional traversal. i.e both forward and backward direction.

- It supports all the four CRUD operations



Declaration of ListIterator

`public interface ListIterator<E> extends Iterator<E>`

Where E represents the generic type i.e any parameter of any type/user-defined object.

Syntax to get a list Iterator on a list

`ListIterator<E> listIterator()`

Methods of ListIterator

ListIterator is a bi-directional iterator. For this functionality, it has two kinds of methods:

1. Forward direction iteration: Returns true if the iteration has more elements.

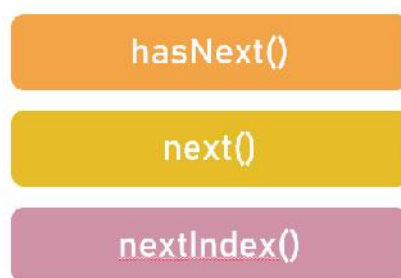


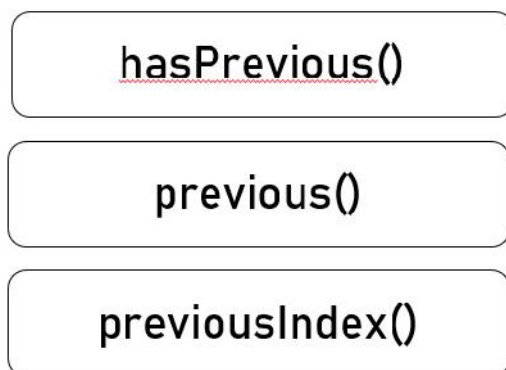
Figure 3:Methods of forwarding direction iteration

`hasNext()` :The `hasNext()` method returns true if there exists an element in the list.

`next()` :The `next()` method returns the next element of the list.

`nextIndex()`: This method returns the index of the element that the `next()` method will return.

Backward direction iterator: With the help of this method we can access the elements in the backward direction.



Methods of backward direction iteration

`previous`: This method returns true if the iteration has more elements while traversing backward.

previous():This method returns the previous element of the list.

previousindex():This method returns the index of the element that the previous() method will return.

- learn the basic concept of LinkedList class.
- understand the different constructors of the LinkedList class.
- analyze the basic methods of the LinkedList class.

3.5 LinkedList Class

LinkedList is a part of the Collection framework present in java.util package. This class is an implementation of the LinkedList data structure which is a linear data structure where the elements are not stored in contiguous locations and every element is a separate object with a data part and address part. The elements are linked using pointers and addresses. Each element is known as a node. Due to the dynamicity and ease of insertions and deletions, they are preferred over the arrays. It also has a few disadvantages like the nodes cannot be accessed directly instead we need to start from the head and follow through the link to reach a node we wish to access. LinkedList class uses a doubly linked list to store the elements. It provides a linked-list data structure. It inherits the AbstractList class and implements List and Deque interfaces.

Declaration

public class LinkedList<E> extends AbstractSequentialList<E> implements List<E>, Deque<E>, Cloneable, Serializable.

The following table illustrates the various LinkedList constructors along with their descriptions.

LinkedList constructors

Constructor	Description
LinkedList()	It is used to construct an empty list.
LinkedList(Collection<? extends E> c)	It is used to construct a list containing the elements of the specified collection, in the order, they are returned by the collection's iterator.

LinkedList Methods

The following table shows the various methods of LinkedList and their usage.

Methods of LinkedList

Method	Description
addFirst()	Adds an item to the beginning of the list.
addLast()	Add an item to the end of the list.
removeFirst()	Remove an item from the beginning of the list.
removeLast()	Remove an item from the end of the list.

getFirst()	Get the item at the beginning of the list.
getLast()	Get the item at the end of the list.

**Example**

```
//Operations Link list
import java.util.LinkedList;

public class LL {
    public static void main(String[] args) {
        LinkedList<String> item = new LinkedList<String>();
        item.add("Pencil");
        item.add("Scale");
        item.add("Pen");
        item.add("Sweets");
        System.out.println(item);
        item.addFirst("Mazda");
        System.out.println(item);
        item.addLast("Mazda");
        System.out.println(item);
        item.removeFirst();
        System.out.println(item);
        item.removeLast();
        System.out.println(item);
        System.out.println(item.getFirst());
        System.out.println(item.getLast());
    }
}
```

Output:

```
PS C:\Users\ OneDrive\Documents\cap615>javac LL.java
```

```
PS C:\Users\ OneDrive\Documents\cap615> java LL
```

```
[Pencil, Scale, Pen, Sweets]
```

```
[Mazda, Pencil, Scale, Pen, Sweets]
```

```
[Mazda, Pencil, Scale, Pen, Sweets, Mazda]
```

```
[Pencil, Scale, Pen, Sweets, Mazda]
```

```
[Pencil, Scale, Pen, Sweets]
```

```
Pencil
```

```
Sweets
```

3.6 TreeSet Class

TreeSet is one of the most notable Java implementations of the SortedSet interface that stores data in a Tree. Whether or not an explicit comparator is provided, a set maintains the order of the components by using their natural ordering. The Java collections framework's TreeSet class implements the capabilities of a tree data structure. It implements the Set interface, which stores data in a tree. It implements the NavigableSet interface and inherits it from the AbstractSet class. The TreeSet class's objects are stored in ascending order.

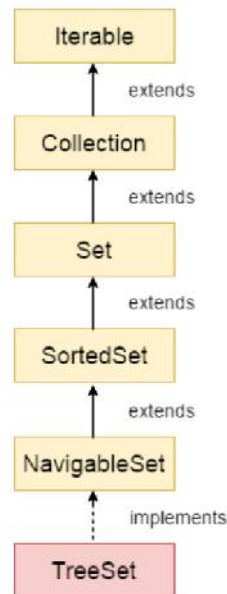


Figure 4: TreeSet Class Hierarchy

The Java TreeSet class implements the NavigableSet interface, as seen in the figure above. In a hierarchical sequence, the NavigableSet interface extends the SortedSet, Set, Collection, and Iterable interfaces.

Important Points

- Like HashSet, the Java TreeSet class only has unique elements.
- The access and retrieval timings for the Java TreeSet class are quite quick.
- Null elements are not allowed in the Java TreeSet class.
- The TreeSet class in Java isn't synchronized.
- The TreeSet class in Java keeps ascending order.

Creating a TreeSet

To make a tree set, we must first import the java.util.TreeSet package. Here's how to make a TreeSet in Java after we've imported the package.

Syntax:

```
TreeSet<Integer> numbers = new TreeSet<>();
```

We've generated a TreeSet without any arguments in this case.

Constructors of TreeSet Class

To make a TreeSet, we must first make an object of the TreeSet class. The TreeSet class contains some constructors that allow the TreeSet to be created. The constructors available in this class are as follows:

TreeSet(): This function creates an empty TreeSet object with members stored in their natural sorting order.

TreeSet(Comparator): This function is used to create an empty TreeSet object with members that require an external sorting order specification.

Syntax: If we want to make an empty TreeSet named TS with external sorting phenomena, we can do so as follows:

```
TreeSet TS= new TreeSet(Comparator comp);
```

TreeSet(Collection): This function creates a TreeSet object that contains all of the elements from the specified collection, with the elements stored in their natural sorting order. In a nutshell, this function is used whenever a conversion from a Collection object to a TreeSet object is required.

Syntax: If we want to make a TreeSet named TS, we can do it by following these steps:

```
TreeSet TS = new TreeSet(Collection coll);
```

TreeSet(SortedSet): This function Object() { [native code] } creates a TreeSet object that contains all of the elements from the given sortedset, with the elements stored in their default natural sorting order. In a nutshell, this function converts a SortedSet object into a TreeSet object.

Syntax: If we want to make a TreeSet named TS, we can do it by following these steps:

```
TreeSet TS = new TreeSet(SortedSetsts);
```

Methods used in TreeSet

The following are the various methods used in the TreeSet Class:

Insert Elements to TreeSet

- add() -It adds the supplied element to the collection
- addAll() -It adds all of the elements from the given collection to the set

Access TreeSet Elements

- The iterator() method can be used to access the items of a tree set.
- We must import the java.util.Iterator package to use this method.

Remove Elements

- remove() - It removes a given element from a collection.
- removeAll() - It removes all of the set's elements.

Navigation Methods

- first() -This method returns the first element of the set
- last() -This method returns the last element of the set
- pollFirst() -It returns and removes the first element from the set
- pollLast() -It returns and removes the last element from the set

higher(), lower(), ceiling(), and floor() () Methods

- higher(element) - Returns the lowest element in the set of elements greater than the supplied element.
- lower(element) - Returns the most significant element among those less than the supplied element.
- ceiling(element) - Returns the element with the lowest value among those that are bigger than the supplied element. It returns the element supplied as an argument if the element passed exists in a tree set.
- floor(element) - Returns the most significant element among those less than the supplied element. It returns the element supplied as an argument if the element passed exists in a tree set.

Methods headSet(), tailSet(), and subSet()

headSet(): All the elements of a tree set before the specified element are returned by the headSet() method.

tailSet(): The tailSet() method retrieves all elements in a tree set that come after the supplied element, including the specified element.

subset(): subSet of (e1, bv1, e2, bv2) All elements between e1 and e2, including e1, are returned by the subSet() function.

The bv1 and bv2 arguments are optional. bv1 has a default value of true, while bv2 has a default value of false.



Example

// Implementation of TreeSet Class

```
import java.util.*;

class TSExample{

    public static void main(String args[]){

        //Creating and adding elements
        TreeSet<String> list=new TreeSet<String>();
        list.add("Good");
        list.add("Great");
        list.add("Welcome");
        list.add("Happy");

        //Traversing elements
        Iterator<String> it=list.iterator();
        while(it.hasNext()){
            System.out.println(it.next());
        }
    }
}
```

Output:

```
PS C:\Users\ OneDrive\Documents\cap615>javac TSExample.java
```

```
PS C:\Users\ OneDrive\Documents\cap615> java TSExample
```

Good

Great

Happy

Welcome

3.7 PriorityQueue class

The PriorityQueue class provides the facility of using a queue. But it does not order the elements in a FIFO manner. It inherits AbstractQueue class. A PriorityQueue is used when the objects are supposed to be processed based on the priority.

Syntax:

```
public class PriorityQueue<E> extends AbstractQueue<E> implements Serializable
```

where E denotes the type of elements in the queue. The class implements Serializable, Iterable<E>, Collection<E>, and Queue<E> interfaces are implemented by this class.

PriorityQueue Insertion and accessing

The following methods are available for inserting elements:

Insertion of elements

- `add()`: Adds the provided element to the queue via the `add()` method. If the queue is full, an exception is thrown.
- `offer()`: Adds an element to the queue with the given name. It returns false if the queue is full.

Accessing elements of the PriorityQueue

The `peek()` method can be used to get elements from a priority queue.

The head of the queue is returned by this procedure.



Example

```
// Accessing of PQ elements
import java.util.PriorityQueue;
class AccessPq {
    public static void main(String[] args) {
        // Creating a priority queue
        PriorityQueue<Integer> num = new PriorityQueue<>();
        num.add(14);
        num.add(12);
        num.add(11);
        System.out.println("PriorityQueue: " + num);
        // Using the peek() method
        int num1 = num.peek();
        System.out.println("Accessed Element: " + num1);
    }
}
```

Output:

```
PS C:\Users\ OneDrive\Documents\cap615\programs> javac AccessPq.java
```

```
PS C:\Users\ OneDrive\Documents\cap615\programs> java AccessPq
```

```
PriorityQueue: [11, 14, 12]
```

```
Accessed Element: 11
```

Removal of elements

We utilize the following methods to remove `PriorityQueue` elements:

- `poll()`: This method returns and removes the head of the queue.
- `remove()`: It removes the specified element from the queue.

Summary

- Because a lot of shifting needs to happen if any element is deleted from the array list, `ArrayList` manipulation is a little slower than `LinkedList` in Java.
- Duplicate elements are possible in the Java `ArrayList` class.

- The poll, remove, peek, and element queue retrieval actions all access the element at the top of the queue.
- The get() method returns the element at the given index, whereas the set() method modifies it.
- An object of type LinkedList<T> represents an ordered sequence of items of type T, but the objects are stored in nodes connected by pointers.
- A TreeSet cannot contain random items since the sorted order of the objects it contains must be determined.
- When a TreeSet compares two objects, it does not use the equals() function. Instead, the compareTo() method is used.
- Consumers can use priority queues to consume higher priority messages first, followed by lower priority ones.
- We can't make a PriorityQueue of non-comparable objects.

Keywords

ArrayList: ArrayList is also known as a resizable array, which can be found in the java util package.

add(): Adds the provided element to the queue via the add() method. If the queue is full, an exception is thrown.

Iterator(): The iterator() method can be used to access the items of a tree set.

poll(): It will give us the latest item and also remove it from the queue.

TreeSet: TreeSet is one of the most notable Java implementations of the SortedSet interface that stores data in a Tree.

first(): If TreeSet is not null, this method will return the first element, otherwise it will throw NoSuchElementException.

Self Assessment

1. Which of these return type of hasNext() method of an iterator?
 - A. Integer
 - B. Double
 - C. Boolean
 - D. Collections Object
2. Which of these methods can be used to move to the next element in a collection?
 - A. next()
 - B. move()
 - C. shuffle()
 - D. hasNext()
3. Which of these iterators can be used only with List?
 - A. Setiterator
 - B. ListIterator
 - C. Iterator
 - D. None of the mentioned

4. Which of the following is/are the methods of ListIterator?

- A. Forward direction iteration
- B. Backward direction iteration
- C. Both
- D. None

5. Iterator and ListIterator can traverse over

- A. lists
- B. sets
- C. maps
- D. All of the above

6. What is the output of this program?

```
import java.util.*;
class Collection_iterators
{
    public static void main(String args[])
    {
        ListIterator a = list.listIterator();
        if(a.previousIndex() != -1)
            while(a.hasNext())
                System.out.print(a.next() + " ");
        else
            System.out.print("EMPTY");
    }
}
```

- A. 0
- B. 1
- C. -1
- D. EMPTY

7. Which of these is a method of ListIterator used to obtain the index of the previous element?

- A. previous()
- B. previousIndex()
- C. back()
- D. goBack()

8. The add and remove methods of TreeSet have a time complexity of:

- A. $O(n)$
- B. $O(n + 1)$
- C. $O(1)$
- D. $O(\log n)$

9. Find the output of the below Java program?

```
import java.util.*;
public class Test {
    public static void main(String[] args) {
        TreeSet<Integer> ts = new TreeSet<Integer>();
        ts.add(100);
        ts.add(0);
        ts.add(150);
        ts.add(50);
        ts.add(125);
        ts.add(125);
        System.out.println(ts);
    }
}
```

- A. [150, 125, 100, 50, 0]
- B. [0, 50, 100, 125, 150]
- C. [100, 0, 150, 50, 125]
- D. [100, 0, 150, 50, 125, 125]

10. What will be the output of the following Java program?

```
import java.util.*;
class Output
{
    public static void main(String args[])
    {
        TreeSet t = new TreeSet();
        t.add("3");
        t.add("9");
        t.add("1");
        t.add("4");
        t.add("8");
        System.out.println(t);
    }
}
```

- A. [1, 3, 5, 8, 9]
- B. [3, 4, 1, 8, 9]
- C. [9, 8, 4, 3, 1]
- D. [1, 3, 4, 8, 9]

11. Which class stores elements in ascending order?

- A. ArrayList
- B. HashSet
- C. TreeSet
- D. All the answers are true

12. Which of the following are the features of TreeSet class?

- A. Java TreeSet class access and retrieval times are quite fast.
- B. Java TreeSet class is non-synchronized.
- C. Java TreeSet class maintains ascending order.
- D. All of the above.

13. PriorityQueue is thread-safe.

- A. True
- B. False
- C. Can't say
- D. May be

14. Which of these classes can generate an array that can increase and decrease in size automatically?

- A. ArrayList()
- B. DynamicList()
- C. LinkedList()
- D. MallocList()

15. Which of this method of the ArrayList class is used to obtain the present size of an object?

- A. size()
- B. length()
- C. index()
- D. capacity()

Answers for Self Assessment

- | | | | | |
|-------|-------|-------|-------|-------|
| 1. C | 2. A | 3. B | 4. C | 5. A |
| 6. D | 7. B | 8. D | 9. B | 10. D |
| 11. C | 12. D | 13. A | 14. A | 15. A |

Review Questions

1. How do you find the number of elements present in an ArrayList give the appropriate example?
2. Explain the various constructors and methods used in TreeSet class.
3. What is the use of ListIterator class? Explain the various methods of ListIterator class.

4. With an example explain the basic operations that we can perform using ArrayList.
5. What are priority queues? Explain the various methods that we can perform on Priority queues?
6. Write a note on the following:
 - a) offer()
 - b) poll()
 - c) peek()
7. Write a program for the implementation of ListIterator class.



Further Readings

Reges, S., &Stepp, M. (2014). *Building Java Programs*. Pearson.

Sedgewick, R. (2002). *Algorithms in Java, Parts 1-4*. Addison-Wesley Professional.

Goodrich, M. T., Tamassia, R., &Goldwasser, M. H. (2014). *Data structures and algorithms in Java*. John Wiley & Sons.

Arnold, K., Gosling, J., & Holmes, D. (2005). *The Java programming language*. Addison Wesley Professional.

Gosling, J., Holmes, D. C., & Arnold, K. (2005). *The Java programming language*.



Web Links

<https://www.freecodecamp.org/news/priority-queue-implementation-in-java/>

<https://www.geeksforgeeks.org/arraylist-in-java/>

<https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html>

<https://www.baeldung.com/java-tree-set>

<https://beginnersbook.com/2013/12/treeset-class-in-java-with-example/>

<https://www.programiz.com/java-programming/priorityqueue>

<https://www.softwaretestinghelp.com/java-priority-queue-tutorial/>

Unit 04: More on Collection Framework

CONTENTS

Objectives

Introduction

4.1 Comparable Interface

4.2 Comparator Interface

4.3 Properties Class

4.4 Lambda Expressions

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

Objectives

- Learn the basic concept of Comparable and Comparator interface.
- Differentiate between Comparable and Comparator interface.
- Know the basic concept of priorities class and Lambda Expressions.
- Understand and implement various constructors and methods of properties class and Lambda Expressions.

Introduction

The Java collections framework is a dependable, high-performance system that allows diverse collections to work together to handle and manage data in the most effective way possible. Dynamic arrays, linked lists, hash tables, trees, and other Java collection interfaces and classes are examples. The framework is structured in such a way that the extension and implementation of collection classes and interfaces to the implementing class are simple.

Each data collection implements an underlying algorithm, such as a linked list, hash set, or tree set, that can be inherited by the child class. There are two sorts of components in the collection framework:

Collection Interfaces: They are abstract data types that are used to represent collections. They make it possible to manipulate collections regardless of their representation specifics. Because Java is an object-oriented programming language, it works primarily through a hierarchical framework.

Collection Classes: They give the collection interfaces concrete implementations. Collection classes are reusable data structures.

4.1 Comparable Interface

Java provides two APIs for sorting objects using class data members:

- Comparable
- Comparator

Programming in JAVA

The user-defined class's objects are ordered using the Java Comparable interface. This interface is part of java.lang package and only has one method, compare To (Object). It only has a single sorting sequence, which means you can only sort the elements by a single data member. It might be anything from a rollno to a name to an age.

A comparable object is capable of comparing itself with another object. The class itself must implements **java.lang.Comparable** interface to compare its instances.

compareTo(Object o) method

It compares the current object to the supplied object. public int compare To(Object o) compares the current object to the specified object.

- If the current object is greater than the requested object, it returns a positive integer.
- If the current object is less than the requested object, the value will be negative.
- If the current object is equal to the requested object, the value is zero.

We can categorize the following elements:

- Objects that are strings
- Objects of the wrapper class
- Class objects that have been defined by the user

The Collections class contains static methods for sorting collection components. We can use TreeSet or TreeMap if the collection items are of the type Set or Map. The elements of the List, on the other hand, cannot be sorted. The Collections class contains methods for sorting List type elements.

public void sort(List list): It's used to sort the List's elements. The Comparable type is required for list elements.



Example:

```
//Implementation of Comparable interface
class Stu implements Comparable<Stu>{
    int roll;
    String name;
    int age;
    Stu(int roll,Stringname,int age){
        this.roll=roll;
        this.name=name;
        this.age=age;
    }
    public int compareTo(Stu s){
        if(roll==s.roll)
            return 0;
        else if(roll>s.roll)
            return 1;
        else
            return -1;
    }
}
// Sorting of elements
```

```
import java.util.*;

public class SortAsc{
    public static void main(String args[]){
        ArrayList<Stu>arrl=new ArrayList<Stu>();
        arrl.add(new Stu(1,"Seerat",34));
        arrl.add(new Stu(13,"Kawar",16));
        arrl.add(new Stu(4,"Deepak",33));
        Collections.sort(arrl);
        for(Stu s:arrl){
            System.out.println(s.roll+" "+s.name+" "+s.age);
        }
    }
}
```

Output

```
PS C:\Users\ OneDrive\Documents\cap615\programs>javac SortAsc.java
```

```
PS C:\Users\ OneDrive\Documents\cap615\programs> java SortAsc
```

```
4 Deepak 33
```

```
13 Kawar 16
```

```
1 Seerat 34
```

4.2 Comparator Interface

A user-defined class's objects are ordered using the Java Comparator interface. This interface is part of java.util package and it has two methods: compare(Object ob1, Object ob2), and equals (Object ele). It supports different sorting sequences, which means you can order the items by any data member, such as rollno, name, age, or anything else.

Unlike Comparable, Comparator is external to the element type we are comparing. It's a separate class. We create multiple separate classes (that implement Comparator) to compare by different members. Collections class has a second sort() method and it takes Comparator. The sort() method invokes the compare() to sort objects. A comparator object can compare two objects belonging to two different classes.

Compare obj1 and obj2 with the following function :

Syntax: public int compare(Object obj1, Object obj2)



Example

// Java program to demonstrate working of Comparator interface

```
import java.io.*;
import java.lang.*;
import java.util.*;

// A class to represent a student.
class Stu {
    int roll;
    String name;
    int age;
```

```

        // Constructor
        public Stu(int roll, String name, int age)
        {
            this.roll = roll;
            this.name = name;
            this.age = age;
        }
        // Used to print student details in main()
        public String toString()
        {
            return this.roll + " " + this.name + " " + this.age;
        }
    }

    class Sortedage implements Comparator<Stu> {
        // Used for sorting in ascending order of roll number
        public int compare(Stu s, Stu s1)
        {
            return s.age - s1.age;
        }
    }

    class Sortedname implements Comparator<Stu> {
        // Used for sorting in ascending order of name
        public int compare(Stu s, Stu s1)
        {
            return s.name.compareTo(s1.name);
        }
    }

    // Driver class
    class AscSort {
        public static void main(String[] args)
        {
            ArrayList<Stu> arrl = new ArrayList<Stu>();
            arrl.add(new Stu(11, "Seerat", 32));
            arrl.add(new Stu(13, "Deepak", 50));
            arrl.add(new Stu(12, "Shallu", 24));
            System.out.println("Before Sorting");
            for (int i = 0; i < arrl.size(); i++)
                System.out.println(arrl.get(i));
            Collections.sort(arrl, new Sortedage());
        }
    }

```

```

System.out.println("\nAfter Sorting");
        System.out.println("\nAge");
        for (int i = 0; i<arrl.size(); i++)
            System.out.println(arrl.get(i));
        Collections.sort(arrl, new Sortedname());
        System.out.println("\nNames");
        for (int i = 0; i<arrl.size(); i++)
            System.out.println(arrl.get(i));
    }
}

```

Output

PS C:\Users\ OneDrive\Documents\cap615\programs>javac AscSort.java

PS C:\Users\ OneDrive\Documents\cap615\programs> java AscSort

Before Sorting

11 Seerat 32

13 Deepak 50

12 Shallu 24

After Sorting

Age

12 Shallu 24

11 Seerat 32

13 Deepak 50

Names

13 Deepak 50

11 Seerat 32

12 Shallu 24

Difference Between Comparable and Comparator Interface

Both Comparable and Comparator are interfaces for sorting collection elements. However, there are numerous distinctions between the Comparable and Comparator interfaces, which are listed below.

Comparable	Comparator
Comparable provides a single sorting sequence . In other words, we can sort the collection based on a single element such as id, name, and price.	The Comparator provides multiple sorting sequences . In other words, we can sort the collection based on multiple elements such as id, name, and price, etc.
Comparable affects the original class i.e., the actual class is modified.	Comparator doesn't affect the original class . i.e., the actual class is not modified.
Comparable provides compareTo() method to sort elements.	Comparator provides compare() method to sort elements.
Comparable is present in java.lang package.	A Comparator is present in java.util package.

Comparable is meant for objects with natural ordering which means the object itself must know how it is to be ordered. For example Roll Numbers of students.	Comparator interface sorting is done through a separate class.
We can sort the list elements of Comparable type by Collections.sort(List) method.	We can sort the list elements of Comparator type by the Collections.sort(List,Comparator) method.

4.3 Properties Class

A permanent set of properties is represented by the Properties class. The Properties can be loaded from or stored in a stream. It's part of the java.util package. The key and value pair are both strings in the properties object. The java.util.Properties class is a Hashtable subclass.

It can be used to calculate the value of a property based on its key. The Properties class has methods for reading and writing data to and from the properties file. It can also be used to obtain a system's attributes. As it is the default, the Properties class can define alternative property lists. The default properties will be searched if a key property is not found in the original Properties list. Multiple threads can share a single Properties object because it does not require external synchronization. The following table shows the constructors of properties class along with their description.



Notes: The Properties class, like its parent class (HashTable), saves data using the hash code of its keys, hence the order in which its components are stored and shown isn't guaranteed.

Constructors of Properties class

Constructor	Description
Properties()	It creates an empty property list with no default values.
Properties(Properties defaults)	It creates an empty property list with the specified defaults.

Methods of Properties Class

The following are the various methods of properties class:

- Properties class to get information from the properties file.



Example

```
// Creation of properties file
user=Good Day
password=Ahead

//Program to read the data from properties file.
import java.util.*;
```

```
import java.io.*;

public class ReadP {

    public static void main(String[] args) throws Exception {
        FileReader reader = new FileReader("db.properties");

        Properties p = new Properties();

        p.load(reader);

        System.out.println(p.getProperty("user"));

        System.out.println(p.getProperty("password"));

    }

}
```

Output

PS C:\Users\ OneDrive\Documents\cap615\programs> javac ReadP.java

PS C:\Users\ OneDrive\Documents\cap615\programs> java ReadP

Good Day

Ahead



Notes: For the creation of property files you have to save the file using the .properties extension.

- Properties class to create the properties file.



Example

// Creation of properties class

```
import java.util.*;
import java.io.*;

public class CreatP {

    public static void main(String[] args) throws Exception {
        Properties p = new Properties();

        p.setProperty("name", "Harseerat");

        p.setProperty("email", "harjinder@gmail.com");

        p.store(new FileWriter("RR.properties"), "First Properties Example");

    }

}
```

Output

PS C:\Users\ OneDrive\Documents\cap615\programs> javac CreatP.java

PS C:\Users\ OneDrive\Documents\cap615\programs> java CreatP

After this a property file is being created with the name RR.

#First Properties Example

#Mon Nov 22 13:49:37 IST 2021

email=harjinder@gmail.com

name=Harseerat

- Properties class to get all the system properties.



Example

```
// To get all System Properties
import java.util.*;
import java.io.*;
public class GetS {
    public static void main(String[] args)throws Exception{
        Properties p=System.getProperties();
        Set set=p.entrySet();
        Iterator itr=set.iterator();
        while(itr.hasNext()){
            Map.Entry entry=(Map.Entry)itr.next();
            System.out.println(entry.getKey()+" = "+entry.getValue());
        }
    }
}
```

Output

```
PS C:\Users\ OneDrive\Documents\cap615\programs>javac GetS.java
PS C:\Users\ OneDrive\Documents\cap615\programs> java GetS
java.runtime.name = Java(TM) SE Runtime Environment
sun.boot.library.path = C:\Program Files\Java\jre1.8.0_181\bin
java.vm.version = 25.181-b13
java.vm.vendor = Oracle Corporation
java.vendor.url = http://java.oracle.com/
path.separator = ;
java.vm.name = Java HotSpot(TM) 64-Bit Server VM
file.encoding.pkg = sun.io
user.country = IN
user.script =
sun.java.launcher = SUN_STANDARD
sun.os.patch.level =
java.vm.specification.name = Java Virtual Machine Specification
user.dir = C:\Users\Harjinder Kaur\OneDrive\Documents\cap615\programs
java.runtime.version = 1.8.0_181-b13
java.awt.graphicsenv = sun.awt.Win32GraphicsEnvironment
java.endorsed.dirs = C:\Program Files\Java\jre1.8.0_181\lib\endorsed
os.arch = amd64
```

4.4 Lambda Expressions

Java SE 8 introduced lambda expressions, which are a novel and important element of the language. It aids in the iteration, filtering, and extraction of data from a collection. A lambda expression is a small piece of code that takes in input and outputs a value.

Lambda expressions are similar to methods, except they don't require a name and can be used directly within a method's body. A single parameter plus an expression make up the most basic lambda expression:

Syntax

parameter -> expression

Wrap several parameters in parenthesis if you're using more than one.



Example

// Program to implement Lambda Expression

```
class LExp
{
    public static void main(String args[])
    {
        // Creating an ArrayList with elements
        // {1, 2, 3, 4}
        ArrayList<Integer>arrL = new ArrayList<Integer>();
        arrL.add(1);
        arrL.add(2);
        arrL.add(3);
        arrL.add(4);

        // Using lambda expression to print all elements of arrL
        arrL.forEach(n ->System.out.println(n));

        // Using lambda expression to print Odd elements of arrL
        arrL.forEach(n ->{ if (n%2 == 0) System.out.println("even" +n); });
    }
}
```

Output

PS C:\Users\ OneDrive\Documents\cap615\programs>javac LExp.java

PS C:\Users\ OneDrive\Documents\cap615\programs> java LExp

```
1
2
3
4
even2
even4
```

Syntax

(parameter1, parameter2) -> expression



Example

// Java program to demonstrate working of lambda expressions with two parameters

```
public class Lexpt
{
```

```

// operation is implemented using lambda expressions
interface FuncInter1
{
    int operation(int a, int b);
}
// sayMessage() is implemented using lambda expressions
interface FuncInter2
{
    void sayMessage(String message);
}
// Performs FuncInter1's operation on 'a' and 'b'
private int operate(int a, int b, FuncInter1 fobj)
{
    return fobj.operation(a, b);
}
public static void main(String args[])
{
    // lambda expression for addition for two parameters
    // data type for x and y is optional.
    // This expression implements 'FuncInter1' interface
    FuncInter1 add = (int x, int y) -> x + y;

    // lambda expression multiplication for two parameters
    // This expression also implements 'FuncInter1' interface
    FuncInter1 multiply = (int x, int y) -> x * y;
    // Creating an object of Test to call operate using
    // different implementations using lambda Expressions
    Lexptobj = new Lexpt();
    // Add two numbers using lambda expression
    System.out.println("Addition is " +
        tobj.operate(10, 3, add));
    // Multiply two numbers using lambda expression
    System.out.println("Multiplication is " +
        tobj.operate(10, 3, multiply));
    // lambda expression for single parameter
    // This expression implements 'FuncInter2' interface
    FuncInter2 fobj = message -> System.out.println("Hello " + message);
    fobj.sayMessage("Good Day");
}
}

```

Output

```
PS C:\Users\ OneDrive\Documents\cap615\programs>javac Lexpt.java
```

```
PS C:\Users\ OneDrive\Documents\cap615\programs> java Lexpt
```

Addition is 13

Multiplication is 30

Hello Good Day

There are a few options for expression. They must return a value immediately and cannot contain variables, assignments, or phrases like if or for. A code block containing curly brackets can be used to do more sophisticated actions. If the lambda expression has to return a value, a return statement should be included in the code block.



Notes: Lambda expressions are just like functions and they accept parameters just like functions.

Points to Remember

- A lambda expression can have zero, one, or more statements in its body.
- Curly brackets are not required when there is only one statement, and the anonymous function's return type is the same as the body expression's.
- When multiple statements are present, they must be enclosed in curly brackets, and the anonymous function's return type must be the same as the type of the value returned within the code block, or void if nothing is returned.

Summary

- The Properties class can also be used to create new properties in the form of key-value pairs, with the key and value both being String objects.
- Properties class is the subclass of Hashtable.
- The Properties class provides methods to get data from the properties file and store data into the properties file. Moreover, it can be used to get the properties of a system.
- The lambda expression should have the same number of parameters as the method, as well as the same return type.
- As a result, a Java lambda expression is a function that can be constructed independently of any class.
- If the lambda expression has to return a value, a return statement should be included in the code block.
- Lambda expressions are similar to methods, except they don't require a name and can be used directly within a method's body.

Keywords

Properties defaults: A default property list linked with a Properties object is stored in this variable.

Comparator: When a user-defined class does not implement the Comparable interface, the Comparator interface is used to compare objects of that class.

Comparable: Each class that implements the Comparable interface must organize its objects in a specific order.

Lambda expressions: A lambda expression is a small piece of code that takes in input and outputs a value.

Programming in JAVA

compare(Object o1, Object o2): This method is used by the comparator interface to compare two objects.

equals(Object O): It's used to compare the current object to the one that's been specified.

Self Assessment

1. What is the value of "emu".compareTo("emu")
 - A. a negative integer
 - B. 0
 - C. a positive integer
 - D. unpredictable

2. If X.compareTo(Y) is positive, then which of the following is true?
 - A. Y.compareTo(X) is positive
 - B. Y.compareTo(X) is negative
 - C. Y.compareTo(X) is zero
 - D. Y.compareTo(X) is unknown

3. The Comparable interface contains which method?
 - A. toCompare
 - B. compare
 - C. compareTo
 - D. compareWith

4. Collection is a(n) _____.
 - A. interface
 - B. class
 - C. framework and interface
 - D. framework and class

5. Which of these interfaces declares the core method that all collections will have?
 - A. set
 - B. EventListner
 - C. Comparator
 - D. Collection

6. In comparator interface the sort() method invokes the to sort objects.
 - A. toCompare()
 - B. compare()
 - C. compareTo()
 - D. compareWith()

7. A comparator object is capable of comparing two objects of two _____ classes.

- A. Different
- B. Same
- C. Both
- D. None

8. _____ is present in java.lang package.

- A. Comparator
- B. Comparable
- C. Both
- D. None

9. Which of the following is/are true about Properties class?

- A. Properties class to get information from the properties file.
- B. Properties class to create the properties file.
- C. Properties class to get all the system properties.
- D. All

10. What is TRUE about Lambda?

- A. It is neither a function nor an interface.
- B. Lambda is denoted with => sign.
- C. Lambda expression enables functions to be passed as an argument.
- D. None of these.

11. How can we write a parameterless Lambda expression?

- A. Pass an empty set of parentheses on the left side of the arrow.
- B. Need to pass curly braces to denote that there is no parameter on the left side of the arrow.
- C. In this particular case, the arrow is not required at all.
- D. No need to pass anything on the left side of the arrow.

12. What is the return type of lambda expression?

- A. String
- B. Object
- C. void
- D. Function

13. Lambdas introduced in Java 8 allow us to process _____

- A. Data as code
- B. Code as data
- C. None of the above
- D. All

14. Lambda expressions in java 8 are based on _____

- A. Procedural programming

- B. Data programming
- C. Functional programming
- D. All

15. Which of these is a class that uses String as a key to store the value in an object?

- A. Array
- B. ArrayList
- C. Dictionary
- D. Properties

Answers for Self Assessment

- | | | | | |
|-------|-------|-------|-------|-------|
| 1. B | 2. B | 3. C | 4. C | 5. D |
| 6. B | 7. A | 8. B | 9. D | 10. C |
| 11. A | 12. D | 13. B | 14. C | 15. D |

Review Questions

1. In Java, how do you utilize comparator and comparable? Give example.
2. Differentiate between comparable and comparator interfaces.
3. With example define the concept of lambda expressions? Explain the different types of lambda expressions.
4. Write a program to implement lambda expression accepting two parameters.
5. What is the use of properties class? Describe the various constructors used in properties class.
6. With example elucidate the various methods of properties class.



Further Readings

Subramaniam, V. (2014). Functional programming in Java: harnessing the power of Java 8 Lambda expressions. Pragmatic Bookshelf.

Sharan, K. (2014). Beginning Java 8 Language Features: Lambda Expressions, Inner Classes, Threads, I/O, Collections, and Streams. Apress.

Warburton, R. (2014). Java 8 Lambdas: Pragmatic Functional Programming. " O'Reilly Media, Inc."

Bloch, J. (2008). Effective java. Addison-Wesley Professional.

Murach, J. (2011). Murach's Java Programming (p. 836). Murach.



Web Links

<https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html>

<https://www.baeldung.com/java-8-comparator-comparing>

<https://java-programming.mooc.fi/part-10/2-interface-comparable>

<https://howtodoinjava.com/java/collections/java-comparable-interface/>

<https://www.scientecheasy.com/2020/12/java-properties.html/>

https://www.w3schools.com/java/java_lambda.asp

<https://www.programiz.com/java-programming/lambda-expression>

Unit 05: Multithreading

CONTENTS

Objectives

Introduction

5.1 Threads

5.2 Creation of Threads

5.3 Life Cycle of Thread

5.4 What is Pooling?

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

Objectives

After this unit you will be able to:

- Learn the basic concept of Multithreading.
- Understand the different ways of implementing Multithreading.
- Understand the various stages in the life cycle of a thread.
- Learn the various methods of implementing thread states.
- Understand the concept of thread communication and the Process of Thread Communication
- Know the difference between wait and sleep.

Introduction

Java programming language has many important and useful concepts. One such concept is the **Thread**. A thread can be referred to as a single sequential flow of control within a program or the unit of execution within a process. A process is normally broken down into tasks and these tasks are further broken down into threads.



Example: Consider the modern operating system, which allows multiple programs to run at once. While typing a document in a system, one can simultaneously listen to music and browse the net. This indicates that the operating system installed in the computer allows multitasking. Similarly, the execution of several processes in a program can also be done simultaneously. Hot Java web browser is an example of such an application, which allows the user to browse a Web page while downloading an image, or playing animations or audio files at the same time.

A thread has a beginning, a sequence of steps for execution, and an end. A thread is not considered as a program, but it runs within a program. Every program contains at least one thread called a primary thread. In Java, the **main()** method is an example of a primary thread.

5.1 Threads

Before we talk about multithreading, let's discuss threads. A thread is a lightweight smallest part of a process that can run concurrently with the other parts (other threads) of the same process. Threads are independent because they all have a separate path of execution that's the reason if an exception occurs in one thread, it doesn't affect the execution of other threads. All threads of a process share a common memory. The process of executing multiple threads simultaneously is known as multithreading.

A process made up of only one thread is called a single-threaded process. A single-threaded process performs only one task at a time whereas, a process having more than one thread, called a multithreaded process, performs different tasks and interacts with other processes at the same time. Figure 1 is the schematic representation of a single-threaded and a multi-threaded process.

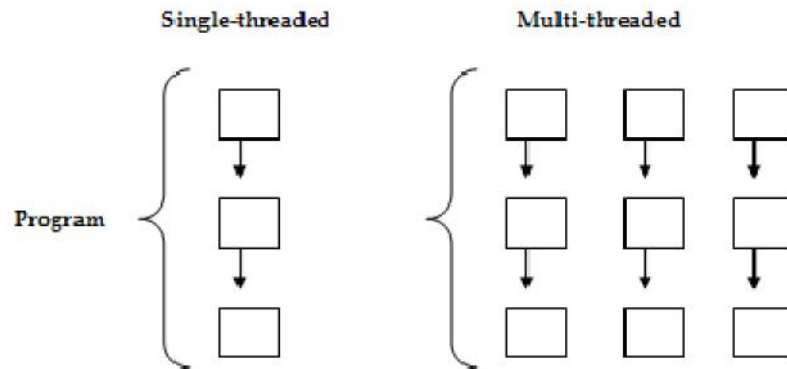


Figure 1: Single-threaded and Multi-threaded Process

In Java, **java.lang.Thread** class creates and controls each thread. A Java program has many threads, and these threads run either asynchronously or synchronously. Multithreading in Java is a process of executing two or more threads simultaneously to maximum utilization of the CPU. Java's multithreading system is built upon the Thread class, its methods, and its companion interface, **Runnable**. Each thread runs parallel to the other.

Multiple threads don't allocate separate memory areas, hence they save memory. It is possible in multithreaded programming to start a new task even when the current task is not completed. Multithreading is conceptual programming, where a program is divided into two or more subprograms, which can be implemented at the same time in parallel. Multithreading is also known as multiple threads of execution. If the application performs many different tasks concurrently, then the threads may access shared data variables to work collaboratively.

Advantages

- The users are not blocked because threads are independent, and we can perform multiple operations at times.
- As such, the threads are independent, the other threads won't get affected if one thread meets an exception.

5.2 Creation of Threads

A thread that is created, must be bound to the **run()** method of an object. At the start of a thread, it invokes the object's **run()** method. Threads can be created by using two mechanisms:

- Extending the Thread class.
- Implementing the Runnable Interface.

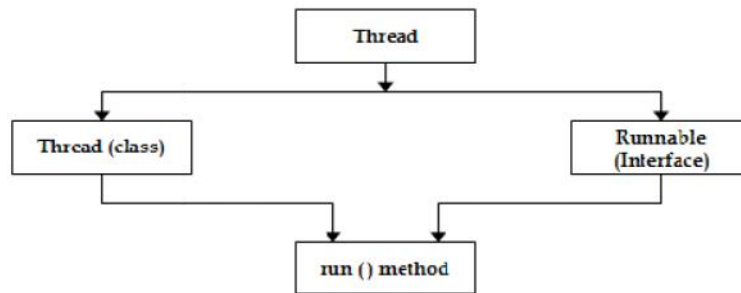


Figure 2: Methods Used for Creation of Threads

Thread creation by extending the Thread class.

The one way used to create a thread is by creating a new class that extends Thread and then creating an instance of that class. The extending class overrides the **run()** method, acting as an entry point for creating a new thread. New threads based on extending the Thread class can be created by following the below-given steps:

1. A class being extended from the Thread class overrides the **run()** method from that Thread, for defining the code to be executed by the thread.

2. This subclass is further called a Thread constructor in its constructors, for initializing the thread. This

is done by using the **super()** call.

3. To make the thread eligible for running, the **start()** method is invoked on the object of the class.



Example :// Code for thread creation by extending the Thread class

```

Class MultithreadingDemo extends Thread {
    public void run()
    {
        try {
            // Displaying the thread that is running
            System.out.println("Thread " + Thread.currentThread().getId() + " is running");
        }
        catch (Exception e) {
            // Throwing an exception
            System.out.println("Exception is caught");
        }
    }
}

// Main Class
public class Multithread {
    public static void main(String[] args)
    {
        int n = 8; // Number of threads
        for (int i = 0; i < n; i++) {
            MultithreadingDemo object
                = new MultithreadingDemo();
            object.start();
        }
    }
}
  
```

```
    }
}
```

Thread creation by implementing the Runnable Interface

We create a new class that implements `java.lang. Runnable` interface and override `run()` method. Then we instantiate a `Thread` object and call the `start()` method on this object. The easiest way to create a thread in Java is by implementing the **runnable** interface and then instantiating an object of that class. Using the **run()** method in the class, is the only method that is used for implementing the **runnable** interface, as this contains the logic of the thread.

Threads based on the **runnable** interface can be created by following the below steps:

1. The **runnable** interface is implemented by a class, providing the **run()** method, which will be executed by the thread. An object of this class type is a **runnable object**.
2. The object of the `Thread` class is developed by passing a **runnable object** as an argument to the `Thread` constructor. Therefore, the `Thread` object now having a **runnable object** implements the **run()** method.
3. The **start()** method is now invoked on the `Thread` object, which is created in the previous step. And, the **start()** method returns immediately after a thread has been produced.
4. The thread ends at the same time when the **run()** method ends.



Example: Java code for thread creation by implementing the `Runnable` Interface

```
class MultithreadingDemo1 implements Runnable {
    public void run()
    {
        try {
            // Displaying the thread that is running
            System.out.println("Thread " + Thread.currentThread().getId() + " is running");
        }
        catch (Exception e) {
            // Throwing an exception
            System.out.println("Exception is caught");
        }
    }
}

// Main Class
class Multithread {
    public static void main(String[] args)
    {
        int n = 8; // Number of threads
        for (inti = 0; i < n; i++) {
            Thread object= new Thread(new MultithreadingDemo());
            object.start();
        }
    }
}
```

5.3 Life Cycle of Thread

A thread in Java at any point of time exists in any one of the following states. A thread lies only in one of the shown states at any instant:

- New
- Runnable
- Blocked
- Waiting
- Timed Waiting
- Terminated

New state

When a new thread is created, it is in a new state. The thread has not yet started to run when a thread is in this state. When a thread lies in the new state, its code is yet to be run and hasn't started to execute.

Runnable State

A thread that is ready to run is moved to a runnable state. In this state, a thread might be running or it might be ready to run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run. A multi-threaded program allocates a fixed amount of time to each thread. Every thread runs for a short while and then pauses and relinquishes the CPU to another thread so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lies in a runnable state.

Blocked/Waiting state

When a thread is temporarily inactive, then it's in one of the following states:

- Blocked: When a thread is waiting for I/O to complete, it lies in the blocked state. It's the responsibility of the thread scheduler to reactivate and schedule a blocked/waiting thread. A thread in this state cannot continue its execution any further until it is moved to a runnable state.
- Waiting: A thread is in the blocked state when it tries to access a protected section of code that is currently locked by some other thread. When the protected section is unlocked, the scheduler picks one of the threads which is blocked for that section and moves it to the runnable state. Whereas, a thread is waiting when it waits for another thread on a condition. When this condition is fulfilled, the scheduler is notified and the waiting thread is moved to a runnable state.

Timed Waiting

A thread lies in a timed waiting state when it calls a method with a time-out parameter. A thread lies in this state until the timeout is completed or until a notification is received.



Example: when a thread calls sleep or a conditional wait, it is moved to a timed waiting state.

Terminated State

A thread terminates because of either of the following reasons:

- Because it exists normally.
- This happens when the code of the thread has entirely executed by the program.
- Because there occurred some unusual erroneous event, like segmentation fault or an unhandled exception.

Implementing the Thread States

In Java, to get the current state of the thread, use **Thread.getState()** method to get the current state of the thread. Java provides **java.lang.Thread.State** class that defines the ENUM constants for the state of a thread.

1. Constant type: New

Declaration:

public static final Thread.State NEW

Description: Thread state for a thread that has not yet started.

2. Constant type: Runnable

Declaration:

public static final Thread.State RUNNABLE

Description: Thread state for a runnable thread. A thread in the runnable state is executing in the Java virtual machine but it may be waiting for other resources from the operating system such as a processor.

3. Constant type: Blocked

Declaration:

public static final Thread.State BLOCKED

Description: Thread state for a thread blocked waiting for a monitor lock. A thread in the blocked state is waiting for a monitor lock to enter a synchronized block/method or reenter a synchronized block/method after calling `Object.wait()`.

4. Constant type: Waiting

Declaration:

public static final Thread.State WAITING

Description: Thread state for a waiting thread. Thread state for a waiting thread. A thread is waiting due to calling one of the following methods:

- `Object.wait` with no timeout
- `Thread.join` with no timeout
- `LockSupport.park` a thread in the waiting state is waiting for another thread to perform a particular action.

5. Constant type: Timed Waiting

Declaration:

public static final Thread.State TIMED_WAITING

Description: Thread state for a waiting thread with a specified waiting time. A thread is in the timed waiting state due to calling one of the following methods with a specified positive waiting time.

6. Constant type: Terminated

Declaration:

public static final Thread.State TERMINATED

Description: Thread state for a terminated thread. The thread has completed execution.



Example: If the loop in `run()` method has fifty iterations, then the life of the thread will be fifty iterations of the loop.

5.4 What is Pooling?

The process of testing a condition repeatedly till it becomes true is known as polling. Java provides the benefits of avoiding thread pooling by using inter-thread communication. Polling is usually implemented with the help of loops to check whether a particular condition is true or not. If it is true, certain actions are taken. This wastes many CPU cycles and makes the implementation inefficient.

To avoid polling, Java uses three methods, namely, `wait()`, `notify()` and `notifyAll()`. All these methods belong to `Object` class as final so that all classes have them. They must be used within a synchronized block only.

- ✓ **wait()** It tells the calling thread to give up the lock and go to sleep until some other thread enters the same monitor and calls notify().
- ✓ **notify()** It wakes up one single thread that is called wait() on the same object. It should be noted that calling notify() does not give up a lock on a resource.
- ✓ **notifyAll()** It wakes up all the threads that are called wait() on the same object.

Inter-thread Communication

Inter-thread communication or **co-operation** is all about allowing synchronized threads to communicate with each other. It is the process in which two threads communicate with each other by using wait (), notify (), and notifyAll () methods. The Thread which is required updation has to call the wait() method on the required object then immediately the Thread will be entered into a waiting state. So, The Thread which is performing the updation of an object is responsible to give notification by calling notify () method. After getting notification the waiting thread will get those updation.

There are three ways for the threads to communicate with each other.

Firstly, all the threads share the same memory space. If the threads share the same object, then these threads share access to that object's data member and thus communicate with each other.

In a second way, threads communicate by using thread control methods. The second way includes

- 1) **suspend()**: By using this method, a thread can suspend itself and wait till another thread resumes it.
- 2) **resume()**: By using this method, a thread can wake up another waiting thread through its resume() method and then run concurrently.
- 3) **join()**: By using this method, the caller thread can wait for the completion of the called thread.

Thirdly, threads communicate by using the following three methods:

- 1) **wait()**: This method tells the calling thread to examine and make the calling thread wait until another calls the same threads to notify() or notifyAll() or a timeout occurs.
- 2) **notify()**: This method wakes only the first waiting thread on the same object.
- 3) **notifyAll()**: This method wakes up all the threads that have been called by wait() on the same object.



Programs for the implementation of wait() and notify().

```
class passenger extends Thread
{
int total = 0;
public void run()
{
synchronized (this)
{
System.out.println("wait .... ");
for (inti = 0; i<= 10; i++)
total = i+2;
System.out.println("passenger is given notification call");
notify();
}
}
```

```

    }
}
public class WaitNotify
{
    public static void main(String[] args) throws InterruptedException
    {
        passenger p = new passenger();

        p.start();
        synchronized (p)
        {
            System.out.println("passenger is waiting for the bus ");
            p.wait();
            System.out.println("passenger got notification");
        }
        System.out.println("after "+p.total+" time");
    }
}

```

Important Points to Remember

1. wait (), notify () and notifyAll () methods are available in the Object class but not in the Thread class because the Thread can call these methods on any common object.
 2. To call wait (), notify () and notifyAll () methods compulsory, the current thread should be the owner of that object.
- Hence, we can call wait(), notify(), and notifyAll() methods only from synchronized area otherwise we will get runtime exception saying IllegalMonitorStateException.
3. Once a thread calls the wait () method on the given object, it releases the lock on that object immediately and enters into the waiting state.
 5. Except for these methods (wait (), notify (), and notifyAll ()), there is no other place or method where the lock releases will happen.
 6. One important thing to note is that when a thread calls wait (), notify (), notifyAll () methods on any object, it releases the lock of that particular object but not all the locks it has.
 7. Also, note that on which object we are calling wait (), notify (), and notifyAll () methods, the corresponding object lock we have to get but no other object locks.



Caution: The three methods wait(), notify(), and notifyAll() must only be called from the synchronized methods.

Inter-thread communication Process

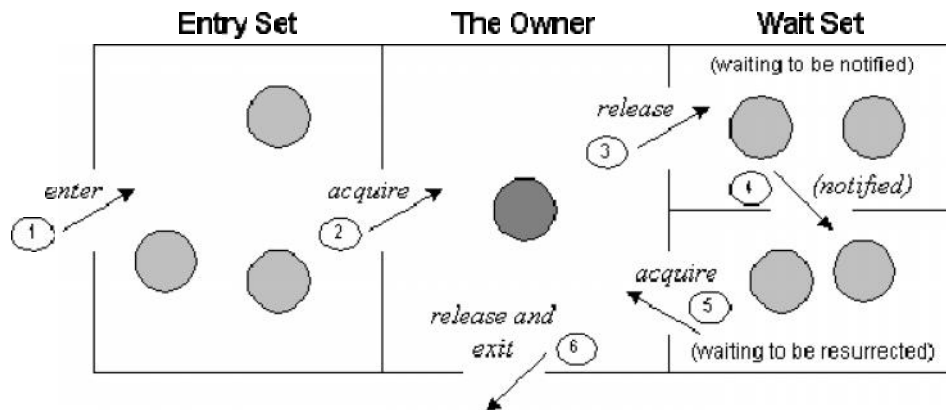


Figure 3: Process of Communication

The point to the point explanation of the above diagram is as follows:

1. Threads enter to acquire a lock.
2. A lock is acquired by one thread.
3. Now thread goes to waitingforthestate if you call the wait() method on the object. Otherwise, it releases the lock and exits.
4. If you call notify() or notifyAll() method, the thread moves to the notified state (runnable state).
5. Now the thread is available to acquire a lock.
6. After completion of the task, the thread releases the lock and exits the monitor state of the object.



Example: Program to illustrate the implementation of **wait()**, **notify()** and **notifyAll()** methods.

class Sort //class **Sort** is created

```
{
int num=0; //integer value num is assigned 0
Boolean value = false; //The variable value is assigned false
synchronized int get() //synchronized block is declared
{
if (value==false) //checking if value is false
try {
wait(); //thread is made to wait
}
catch (InterruptedException e) {
System.out.println("InterruptedException caught");
}
System.out.println("consume: " + num);
value=false;
notify(); //notify method is invoked
return num; //returns num value
}
```

```

synchronized void put(intnum)
{
    if (value==true) // checking if value is true
    try {
        wait(); //thread is made to wait
    }
    catch (InterruptedException e) {
        System.out.println("InterruptedException caught");
    }
    this.num=num; //num value points to the current class object
    System.out.println("Produce: " + num);
    value=false;
    notify(); //notify method is invoked
}
}

//class Construct is declared using the keyword extends
class Construct extends Thread
{
    Sort s; //sort is declared with object s (Synchronized method)
    Construct(Sort s)
    {
        this.s=s; //s points to the current class objects
        this.start();
    }
    public void run() //run() method is invoked
    {
        inti=0; //integer i is assigned to value 0
        s.put(++i); //increments the value num
    }
}

class User extends Thread //class User is declared using extends keyword
{
    Sort s; //sort is declared with object
    User(Sort s) //constructor is called with Sort s (Synchronized method)
    {
        this.s = s; //s points to the current class objects
        this.start();
    }
    public void run() {
        s.get(); //retrieves the produced number and returns it to the output
    }
}

```

```

}
public class InterThread //class InterThread is declared
{
public static void main(String[ ] args)
{
Sort s=new Sort();
new Construct(s);
new User(s);
}
}

```

Output:

C:\nisha>javac InterThread.java

C:\nisha>java InterThread

Produce: 1

consume: 1

In this program, two threads **Construct** and **User** share the synchronized methods of the class **Sort**. At the time of program execution, the **put()** method of the **Construct** class is invoked, which increments the variable **num** by 1. After producing 1 by the **Construct**, the method **get()** is invoked by the **User** class, which retrieves the produced number and returns it to the output. Thus, the **user** cannot retrieve the number without producing it.

Difference between wait and sleep

wait()	sleep()
wait() method releases the lock.	sleep() method doesn't release the lock.
is the method of Object class.	is the method of Thread class.
is the non-static method.	is the static method.
is the non-static method.	is the static method.
should be notified by notify() or notifyAll() methods.	after the specified amount of time, sleep is completed.

**Lab Exercise**

1. Write a program to create multiple threads.
2. Write a program that checks whether a given number is a prime using both the Thread class and Runnable Interface.

Summary

- A thread can be referred to as a sequential flow of control within a program or the unit of execution within a process.
- The Thread class is defined in the pre-defined package **java.lang**, which needs to be imported into the program code, so that our classes are aware of their definition.

- Multithreading is the ability of an operating system to execute different parts of a program simultaneously.
- There are four possible states in the life cycle of a thread, namely, the **new thread** state, the **runnable** state, the **not runnable** state, and the **dead** state.
- There are two ways to create a thread. First, by implementing the **Runnable** interface, and second, by extending the Thread class.
- A class being extended from the Thread class overrides the **run()** method from that Thread, for defining the code to be executed by the thread.
- **wait()**, **notify()** and **notifyAll()** methods are available in the Object class but not in the Thread class because the Thread can call these methods on any common object.

Keywords

Thread: A thread is a lightweight sub-process, the smallest unit of processing.

Multithreading: Multithreading refers to a process of executing two or more threads simultaneously for maximum utilization of the CPU.

run(): Method used to implement the code that needs to be executed by our thread.

start(): Method that causes the thread to move into the **Runnable** state.

suspend(): By using this method, a thread can suspend itself and wait till another thread resumes it.

resume(): By using this method, a thread can wake up another waiting thread through its **resume()** method and then run concurrently.

join(): By using this method, the caller thread can wait for the completion of the called thread.

wait(): This method tells the calling thread to examine and make the calling thread wait until another calls the same threads to **notify()** or **notifyAll()** or a timeout occurs.

notify(): This method wakes only the first waiting thread on the same object.

notifyAll(): This method wakes up all the threads that have been called by **wait()** on the same object.

Self Assessment

1. What does sleep in Thread class do?
 - A. Causes the thread, which sleep is invoked on, to sleep (temporarily cease execution) for the specified number of milliseconds
 - B. Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds
 - C. Causes the main() thread to sleep for the specified number of milliseconds
 - D. Causes the currently executing thread to wait (temporarily cease execution) for
2. Which state does the thread enter, when a thread class is created at any instance?
 - A. New thread
 - B. Runnable
 - C. Not Runnable
 - D. Dead
3. What is multithreaded programming?

- A. It's a process in which two different processes run simultaneously
 - B. It's a process in which two or more parts of the same process run simultaneously
 - C. It's a process in which many different processes can access the same information
 - D. It's a process in which a single process can access information from many sources
4. Which of these statements is incorrect?
- A. By multithreading CPU idle time is minimized, and we can make maximum use of it
 - B. By multitasking CPU idle time is minimized, and we can make maximum use of it
 - C. Two threads in Java can have the same priority
 - D. A thread can exist only in two states, running and blocked
5. Execution of a java thread begins on which method call?
- A. Start ()
 - B. Run ()
 - C. Execute ()
 - D. Launch ()
6. A process can have
- A. only one thread
 - B. one or multiple threads
 - C. multiple sub-threads
 - D. multiple sub-threads & thread
7. We can create a thread in java by
- A. implementing Thread
 - B. extending Thread
 - C. extending Runnable
 - D. All of the above
8. The process of executing multiple threads simultaneously is termed as _____
- A. Multiprocessing
 - B. Multipurpose
 - C. Multithreading
 - D. All of the above
9. Which of these methods is used to find out that a thread is still running or not?
- A. run()
 - B. Alive()
 - C. isAlive()
 - D. checkout()
10. Which of these interfaces is implemented by the Thread class?
- A. Runnable
 - B. Connections

- C. Set
- D. MapConnections

11. Which of these classes is used to make a thread?

- A. String
- B. System
- C. Thread
- D. Runnable

12. Thread priority in Java is?

- A. Integer
- B. Float
- C. double
- D. long

13. The wait(), notify(), and notifyAll() methods are present in which class or interface?

- A. Object class
- B. Thread class
- C. Runnable class
- D. None of these

14. The wait(), notify(), and notifyAll() methods can be called from?

- A. Only Synchronized Method
- B. All Synchronized block/method
- C. Only Static synchronized block/method
- D. Only Synchronized Block

15. When a thread calls the wait() method, which of the following options is valid?

- A. Immediately the thread will enter into the waiting state without releasing any lock.
- B. A thread will release the lock of that object but may not immediately.
- C. A thread will release all acquired locks and immediately enter into the waiting state.
- D. A thread will immediately release the lock of that particular object and enter into the waiting state.

Answers for Self Assessment

- | | | | | |
|-------|-------|-------|-------|-------|
| 1. B | 2. A | 3. B | 4. D | 5. A |
| 6. B | 7. B | 8. C | 9. C | 10. A |
| 11. C | 12. A | 13. A | 14. B | 15. D |

Review Questions

1. "A thread that is created, must be bound to the run() method of an object". Comment.
2. "When a thread is alive, it indicates it is in one of its several states". Justify.
3. "The second way that is used to create a thread is, by creating a new class that extends Thread, and then creating an instance of that class". Elaborate.
4. "Understanding the life cycle of a thread is very important, especially at the time of developing codes using threads". Elaborate.
5. What are the different states of a thread, or what is thread lifecycle?
6. With an example explain the different methods of creating a User thread?
7. "There are three ways for the threads to communicate with each other". Elaborate.



Further Readings

E Balagurusamy, Programming with Java_A Primer 3e, New Delhi

Herbert Schildt, The Complete Reference, 7th edition, Tata McGraw Hill

Horton, I. (2005). Ivor Horton's Beginning Java 2. John Wiley & Sons.

Lewis, B., & Berg, D. J. (2000). *Multithreaded programming with Java technology*. Prentice Hall Professional.

Arnold, K., Gosling, J., & Holmes, D. (2005). *The Java programming language*. Addison Wesley Professional.



Web Links

<https://www.javatpoint.com/multithreading-in-java>

<https://www.geeksforgeeks.org/multithreading-in-java/>

<https://www.mygreatlearning.com/blog/multithreading-in-java/>

<https://www.scientecheasy.com/2020/08/inter-thread-communication-in-java.html/>

<https://dotnettutorials.net/lesson/inter-thread-communication-in-java/>

Unit 06: More on Multithreading

CONTENTS

Objectives

Introduction

6.1 Suspending a Thread

6.2 Resuming a Thread

6.3 Deadlock

6.4 Stopping a Thread

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

Objectives

After this unit you will be able to:

- Learn the basic concept of suspending and resuming a thread.
- Implement `suspend()` and `resume()`.
- Know the basic concept of Deadlock.
- Understand the various methods of stopping a thread.
- Implementation of a deadlock situation and its solution.

Introduction

In Java, a simple program may contain many threads. Each thread may perform different tasks. Sometimes, it becomes necessary to suspend the execution of a thread for a while. This can be done by using `suspend()` method of the class `Thread`. The time needs to be specified till the thread remains suspended and then we can restart the thread by using the `resume()` method of the class `Thread`. If a thread is suspended, it can be restarted. But if a thread is stopped by using the `stop()` method, it cannot be restarted again. These methods were used in the earlier systems of Java, but they are not used in the latest version because sometimes `suspend()` and `stop()` methods of the class `Thread` cause system failure. Therefore, the `run()` method is used in the latest version of Java instead of the above-mentioned methods. So, the `run()` method checks when a thread should be suspended, resumed, or stopped.

6.1 Suspending a Thread

Due to the multi-threading concept, we may need to block one particular thread while some other thread is executing.

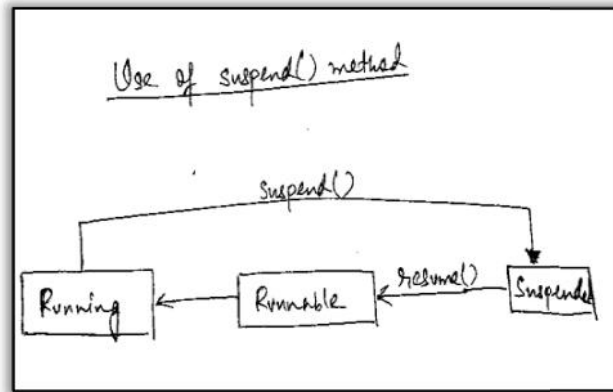


Figure 1: Life Cycle of Thread

As we have seen in the life cycle diagram, there are several inbuilt methods provided to us by the thread class to change the state of a thread. One of them is the `suspend()` method. It is used to change the thread from a Running state to a Suspended/Blocked state.



Example

//Program for the implementation of suspending method

```

public class Testing extends Thread
{
    public void run()
    {
        try
        {
            for(int i=0;i<7;i++)
            {
                Thread.sleep(500);
                System.out.println(this.getName() + ": " + i);
            }
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }

    public static void main(String args[])
    {
        Testing srd1=new Testing();
        Testing srd2=new Testing();
        srd1.setName("First");
        srd2.setName("Second");
        srd1.start();
        srd2.start();
    }
}
  
```

```
try
{
Thread.sleep(1000);
srd1.suspend();
System.out.println("Suspending thread First");
Thread.sleep(1000);
srd1.resume();
System.out.println("Resuming thread First");
Thread.sleep(1000);
srd2.suspend();
System.out.println("Suspending thread Second");
Thread.sleep(1000);
srd2.resume();
System.out.println("Resuming thread Second");
}
catch (InterruptedException e)
{
e.printStackTrace();
}
}
```

Output :

PS C:\Users\OneDrive\Documents\cap615\programs>javac Testing.java

PS C:\Users\OneDrive\Documents\cap615\programs> java Testing

Second: 0

First: 0

Suspending thread First

Second: 1

Second: 2

Resuming thread First

First: 1

Second: 3

First: 2

Second: 4

Suspending thread Second

First: 3

First: 4

Resuming thread Second

Second: 5

First: 5

Second: 6

First: 6

6.2 Resuming a Thread

Suspend() method puts the thread from running to waitingforthestate. And thread can go from waiting to runnable state only when the resume() method is called on the thread. Suspend method is a deprecated method. The resume () method is only used with suspend() method that's why it's also a deprecated method. Suspend() and remove() are deprecated methods because if not used properly they might lead to deadlock.



Example

```
// Implementation of resume() method
public class Resume implements Runnable {

    public void run() {
        for (inti = 0; i<3; i++) {
            System.out.println(Thread.currentThread().getName() + " " + i);
        }
    }

    public static void main(String args[]) throws Exception {
        Thread th = new Thread(new Resume());
        Thread th1 = new Thread(new Resume());
        System.out.println("Starting " + th.getName() + "...");
        th.start();
        System.out.println("Suspending " + th.getName() + "...");
        //Suspend the thread.
        th.suspend();
        System.out.println("Starting " + th.getName() + "...");
        th1.start();
        th1.join();
        // Resume the thread.
        th.resume();
        System.out.println("Starting " + th.getName() + "...");
    }
}
```

Output:

```
PS C:\Users\OneDrive\Documents\cap615\programs>javac Resume.java
```

```
PS C:\Users\ OneDrive\Documents\cap615\programs> java Resume
```

```
Starting Thread-0...
```

```
Suspending Thread-0...
```

```
Starting Thread-0...
```

```
Thread-1 0
```

```
Thread-1 1
```

Thread-1 2

Starting Thread-0...

Thread-0 0

Thread-0 1

Thread-0 2



Caution: Do not create too many threads, as it may consume more CPU time than executing the program.

6.3 Deadlock

Deadlock describes a situation where two or more threads are blocked forever, waiting for each other. A Java multithreaded program may suffer from the deadlock condition because the synchronized keyword causes the executing thread to block while waiting for the lock, or monitor, associated with the specified object. The synchronized keyword is used to make the class or method thread-safe which means only one thread can have a lock of synchronized method and use it, other threads have to wait till the lock releases and anyone of them acquire that lock. It is important to use if our program is running in a multi-threaded environment where two or more threads execute simultaneously. But sometimes it also causes a problem which is called **Deadlock**.

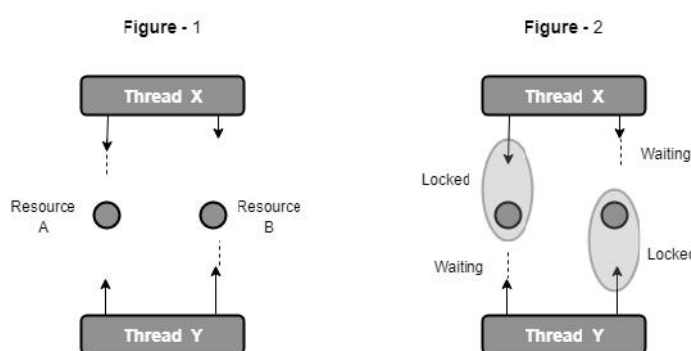


Figure 2: Deadlock Situation



Example: One thread enters the monitor on object A, and another thread enters the monitor on object B. If the thread in A tries to call any synchronized method on B, it becomes blocked. But, if the thread in B tries to access any synchronized method on A, then the thread waits for a while since it should release its lock on B so that the first thread can complete.

Deadlock is considered as an error, which is difficult to debug because of two reasons:

1. It occurs rarely when two threads time-slice in the correct way.
2. It may have more than two synchronized objects and two threads.



Example

//Program to implement deadlock.

```
public class DeadLock
{
    public static void main(String[] args)
    {
        final String rs1 = "Group1";
        final String rs2 = "Group2";
```

```
Thread t1 = new Thread()
{
public void run()
{
synchronized (rs1)
{
System.out.println("Thread 1: locked resource 1");
try
{
Thread.sleep(50);
}
catch (InterruptedException e)
{
}
}
synchronized (rs2)
{
System.out.println("Thread 1: locked resource 2");
}
}
};

Thread t2 = new Thread()
{
public void run()
{
synchronized (rs1)
{
System.out.println("Thread 2: locked resource 2");
try
{
Thread.sleep(50);
}
catch (InterruptedException e)
{
}
}

synchronized (rs2)
{
System.out.println("Thread 2: locked resource 1");
}
}
```

```

    }
    };
    t1.start();
    t2.start();
    }
}

```

Output:

```
PS C:\Users\ OneDrive\Documents\cap615\programs>javac DeadLock.java
```

```
PS C:\Users\ OneDrive\Documents\cap615\programs> java DeadLock
```

Thread 1: locked resource 1

Thread 1: locked resource 2

Thread 2: locked resource 2

Thread 2: locked resource 1

Deadlock Solution

Avoid Unnecessary Locks: We can have a lock for only those members who are required. Having a lock unnecessarily can lead to a deadlock.

Avoid Nested Locks: A deadlock mainly happens when we give locks to multiple threads. Avoid giving a lock to multiple threads if we already have given to one.

Using Thread.join() Method: A deadlock condition appears when one thread is waiting for another to finish. If this condition occurs we can use Thread.join() with the maximum time the execution will take.

Use Lock Ordering: Always assign a numeric value to each lock. Before acquiring the lock with a higher numeric value, acquire the locks with a lower numeric value.

Lock Time-out: We can also specify the time for a thread to acquire a lock. If a thread does not acquire a lock, the thread must wait for a specific time before retrying to acquire a lock.

Just changing the order of the locks prevent the program from going into a deadlock situation



Task: Write a program that displays the name of the thread that executes the main method.

6.4 Stopping a Thread

A thread is automatically destroyed when the run() method has been completed. But it might be required to stop a thread before it has completed its life cycle.

Modern ways to suspend/stop a thread are :

- Using a boolean flag
- Thread.interrupt() method.



Example

// Implementation of stopping a thread using a boolean flag

```
classMyThread implements Runnable {
```

```
    // to stop the thread
```

```
    privatebooleanexit;
```

```
    private String name;
```

```

    Thread t;
    MyThread(String threadname)
    {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        exit = false;
        t.start(); // Starting the thread
    }
    // execution of thread starts from run() method
    public void run()
    {
        inti = 0;
        while (!exit) {
            System.out.println(name + ": " + i);
            i++;
            try {
                Thread.sleep(100);
            }
            catch (InterruptedException e) {
                System.out.println("Caught:" + e);
            }
        }
        System.out.println(name + " Stopped.");
    }
    // for stopping the thread
    public void stop()
    {
        exit = true;
    }
}
// Main class
public class MainB {
    public static void main(String args[])
    {
        // creating two objects t1 & t2 of MyThread
        MyThread t1 = new MyThread("First thread");
        MyThread t2 = new MyThread("Second thread");
        try {
            Thread.sleep(500);
            t1.stop(); // stopping thread t1
            t2.stop(); // stopping thread t2
        }
    }
}

```

```

Thread.sleep(500);
    }
catch (InterruptedException e) {
System.out.println("Caught:" + e);
    }
System.out.println("Exiting the main Thread");
    }
}

```

Output:

PS C:\Users\ OneDrive\Documents\cap615>javac MainB.java

PS C:\Users\ OneDrive\Documents\cap615> java MainB

New thread: Thread[First thread,5,main]

New thread: Thread[Second thread,5,main]

First thread: 0

Second thread: 0

Second thread: 1

First thread: 1

First thread: 2

Second thread: 2

Second thread: 3

First thread: 3

First thread: 4

Second thread: 4

First thread Stopped.

Second thread Stopped.

Exiting the main Thread



Example:

// Implementation of stopping a thread using Thread.interrupt() method

class MyThread implements Runnable {

Thread t;

MyThread()

{

t = new Thread(this);

System.out.println("New thread: " + t);

t.start(); // Starting the thread

}

// execution of thread starts from run() method

public void run()

{

```

while (!Thread.interrupted()) {
    System.out.println("Thread is running");
}
System.out.println("Thread has stopped.");
}
}
// Main class
public class Main {
    public static void main(String args[])
    {
        // creating objects t1 of MyThread
        MyThread t1 = new MyThread();
        try {
            Thread.sleep(1);
            // t1 is an object of MyThread
            // which has an object t
            // which is of type Thread
            t1.t.interrupt();
            Thread.sleep(5);
        }
        catch (InterruptedException e) {
            System.out.println("Caught: " + e);
        }
        System.out.println("Exiting the main Thread");
    }
}

```

Output

```

PS C:\Users\ OneDrive\Documents\cap615>javac MainA.java
PS C:\Users\ OneDrive\Documents\cap615> java MainA
New thread: Thread[Thread-0,5,main]
Thread is running
Thread is running
Thread is running
Thread is running
Thread has stopped.
Exiting the main Thread

```

Summary

- Sometimes, it is necessary to suspend the execution of a thread. This can be done by suspend() method.
- Thread can be restarted by using the resume() method. Thread can be stopped by using the stop() method.

- Deadlock describes a situation where two or more threads are blocked forever, waiting for each other.
- The **resume()** method of thread class is only used with **suspend()** method. This method is used to resume a thread that was suspended using **suspend()** method.
- If a thread is suspended, it can be restarted. But if a thread is stopped by using the **stop()** method, it cannot be restarted again.
- The **synchronized** keyword is used to make the class or method thread-safe which means only one thread can have a lock of synchronized method and use it, other threads have to wait till the lock releases and anyone of them acquire that lock.
- Deadlock is caused because of the **synchronized** keyword because it blocks the executing thread to wait for the lock.
- A **synchronized** block ensures that only one thread executes inside it at a time.

Keywords

suspend(): This method puts a thread in the suspended state and can be resumed using the **resume()** method.

synchronized: **synchronized** keyword is used to make the class or method thread-safe. The keyword ensures that only one thread can access the resource at a given point in time.

stop(): The **stop()** method of the Thread class terminates the thread execution. Once a thread is stopped, it cannot be restarted by the **start()** method.

resume() : The **resume()** method is also deprecated. It does not cause problems, but cannot be used without the **suspend()** method as its counterpart.

Self Assessment

1. By using which of the following method can the suspend thread be revived?
 - A. **suspend()** method
 - B. **start()** method
 - C. **resume()** method
 - D. **end()** method
2. Select a correct answer:
 - A. Stop the running thread by calling the **wait()** method.
 - B. Stop the running thread by calling the **suspend()** method.
 - C. Stop the running thread by calling the **sleep()** method.
 - D. The thread stopped by calling the **stop()** method.
3. The method of the thread is called before the method and carries out any initialization.
 - A. **suspend, resume**
 - B. **start, run**
 - C. **start, stop**
 - D. **resume, suspend**

4. Which method is used to suspend threads that don't need to run when the applet is not visible?
- A. destroy()
 - B. paint()
 - C. stop()
 - D. start()
5. What are valid statements for suspend() and resume() method?
- A. Suspend() method is deadlock prone.
 - B. If the target thread holds a lock on an object when it is suspended, no thread can lock this object until the target thread is resumed.
 - C. If the thread that would resume the target thread attempts to lock this monitor before calling resume, it results in deadlock formation.
 - D. All
6. Which method can make Thread go from running to waitingforthestate?
- A. wait()
 - B. sleep()
 - C. suspend()
 - D. All
7. _____ is used to change the thread from a Running state to Blocked state.
- A. suspend() method
 - B. start() method
 - C. resume() method
 - D. end() method
8. What should not be done to avoid deadlock?
- A. Avoid using multiple threads
 - B. Avoid holding several locks at once
 - C. Execute foreign code while holding a lock
 - D. Use interruptible locks
9. Deadlock situations occur because of which keyword?
- A. this
 - B. synchronized
 - C. super
 - D. static
10. Deadlock is a situation when the thread is waiting for another thread to release the acquired object.
- A. TRUE
 - B. FALSE
 - C. Can be true or false

D. can not say

11. Which of the following method/methods provide a solution to the deadlock.

- A. Avoid Unnecessary Locks
- B. Avoid Nested Locks
- C. Using Thread.join() Method
- D. All of the above

12. A thread is automatically destroyed when the _____ method has completed.

- A. run()
- B. wait()
- C. notify()
- D. suspend()

13. Which of the following methods will cause a thread to stop?

- A. Invoking the interrupt() of the thread
- B. Invoking the sleep() method on thread
- C. When execution of the run() method ends
- D. None of the above

14. Which of the following will directly stop the execution of a Thread?

- A. notify()
- B. notifyall()
- C. wait()
- D. exits synchronized code

15. To kill a thread at any stage..... the method is used.

- A. kill()
- B. stop()
- C. destroy()
- D. dead()

Answers for Self Assessment

- | | | | | |
|-------|-------|-------|-------|-------|
| 1. C | 2. B | 3. B | 4. C | 5. D |
| 6. D | 7. A | 8. C | 9. B | 10. A |
| 11. D | 12. A | 13. C | 14. C | 15. B |

Review Questions

1. "Deadlock is a special type of error that needs to be avoided in multitasking". Elaborate.

2. “Java’s synchronized keyword ensures that only one thread at a time is in a critical region”.Comment.
3. “Each thread may perform different tasks. Sometimes, it becomes necessary to suspend the execution of a thread for a period of time”. Comment.
4. How we can resume from a suspended thread? Explain with the help of an example?
5. With an example explain the different methods of stopping a thread.
6. Explain the different methods used for avoiding deadlocks.



Further Readings

Herbert Schildt, The Complete Reference, 7th edition, Tata McGraw Hill

D. Samanta, Object-Oriented Programming with C++ and Java

Carver, R. H., & Tai, K. C. (2005). Modern multithreading: implementing, testing, and Debugging Multithreaded Java and C++/Pthreads/Win32 programs. John Wiley & Sons.

Lewis, B., & Berg, D. J. (2000). Multithreaded programming with Java technology. Prentice-Hall Professional.

Horstmann, C. S., & Cornell, G. (2001). Core Java 2: Fundamentals (Vol. 1). Prentice-Hall Professional.



Web Links

<http://www.herongyang.com/Java/Deadlock-What-Is-Deadlock.html>

http://java.sun.com/docs/books/jls/second_edition/html/memory.doc.html

http://tim.oreilly.com/pub/a/onjava/excerpt/jthreads3_ch6/index1.html

<https://www.edureka.co/blog/deadlock-in-java/>

<https://www.javatpoint.com/deadlock-in-java>

<https://medium.com/javarevisited/deadlock-in-java-multi-threading-627aaaa0e02>

Unit 07: Synchronization

CONTENTS

Objective

Introduction

7.1 Thread Synchronization

7.2 Exception Handling

7.3 Exception Handling During Multithreading

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

Objective

- Understand the basic concept of thread synchronization and its declaration.
- Implementation of thread synchronization and Exception handling in a multithreaded environment.
- Know the basics of Exception Handling and implementation in multithreaded environment.

Introduction

Threads run concurrently and are independent of each other. This indicates that the threads run in their own space, without being concerned about the status and activities of the other threads that are running concurrently. During this process, threads do not require any method or outside resources, and as a result, threads do not communicate with each other. These types of threads are generally called asynchronous threads. Sharing the same resource (variable/method) by two or more threads is the major problem suffered by asynchronous threads, as only one thread can access a resource at a time. Let us assume that two threads Thread1 and Thread2 are the producers and the consumer processes respectively. These two threads share the same data. A situation can arise in which either the producer produces the data faster than it is consumed or the consumer retrieves the data faster than it is produced. This problem is schematically represented in Figure 1.

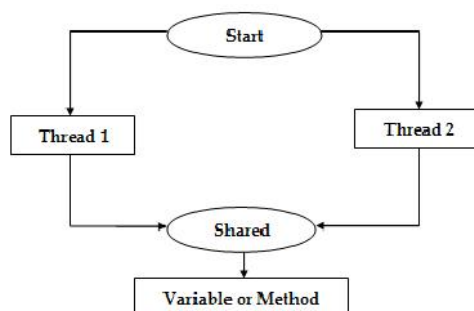


Figure 1: Problem of Sharing the Same Program's Data

To avoid and solve the above problem, Java uses a monitor, which is commonly known as a semaphore. This prevents the data from being corrupted by multiple threads. We can implement

monitor by using the keyword `synchronized` to synchronize threads so that they can intercommunicate with each other. This mechanism allows two or more threads to share the resources sequentially. Java's `synchronized` keyword ensures that only one thread at a time is in a critical region. This region is a locked area, where only one thread at a time is run (or locked). Therefore, once the thread is in its critical region, no other thread can enter into that critical region, and the thread has to wait until and unless the current thread leaves its critical region.

7.1 Thread Synchronization

Synchronization is the ability to manage several threads' access to a common resource. When we only want one thread to access a shared resource, this is the superior alternative. Using synchronized blocks, Java allows you to create threads and synchronize their tasks. The `synchronized` keyword in Java is used to identify synchronized blocks. In Java, a synchronized block is synchronized on an object. Only one thread can execute at a time inside all synchronized blocks synchronized on the same object. All additional threads attempting to enter the synchronized block are prevented until the thread currently inside the block exits.

The syntax for Synchronizing an Object

```
synchronized (<object>)
{
//statements to be synchronized
}
```

By using the above syntax, the methods of the object can only be invoked by one thread at a time.

Monitors are used to implement this synchronization. At any given time, only one thread can own a monitor. A thread is considered to have entered the monitor when it gains a lock. All subsequent threads attempting to access the locked monitor will be suspended until the initial thread leaves.

There are two ways, in which, the synchronization of the execution of code can be done. These two ways are:

1. Synchronized Methods
2. Synchronized Blocks (Statements)

Synchronized Methods

If any method is specified with the keyword `synchronized`, it will be executed by only one thread at a time. For any thread to execute the synchronized method, first, it has to obtain the object's lock. However, if the lock is held by another thread, then the calling thread has to wait. These methods are useful in situations where different methods need to be executed concurrently so that these methods can intercommunicate and manipulate the state of an object.



Example

```
//Program to illustrate the synchronized method.
class share extends Thread //class share is declared using the keyword extends
{
static String msg[]={"Following", "is", "a", "synchronized", "variable"}; //msg is
declared as static
share(String threadname)
//share is called with the parameter threadname
{
super(threadname); //super class is declared
}
public void run( ) //run( ) method is invoked
{
display(getName( )); //displays the name of the thread
```

```

}
public synchronized void display(String threadN)
{
for(int k=0; k<=4; k++)
System.out.println(threadN+msg[k]);
try
{
this.sleep(1500); //thread is made to sleep
} catch(Exception e) { }
}
}
}
public class SynchroThread1
{
public static void main(String[ ] args)
{
share T1=new share("Thread 1:"); //a new thread is created
T1.start( ); //thread is started
share T2=new share("Thread 2:"); //a new thread is created
T2.start( );//thread is started
}
}

```

Output

```

SynchroThread1
Thread 1: Following
Thread 1: is
Thread 1: a
Thread 1: synchronized
Thread 1: variable
Thread 2: Following
Thread 2: is
Thread 2: a
Thread 2: synchronized
Thread 2: variable

```



Task: Write a program to illustrate the synchronization of the print method that displays the following output:
 My name is Rahul
 I am working in ABC Company.

Synchronized Blocks (Statements)

An alternative way of handling synchronization is by using Synchronized Blocks (Statements).



Example

// Program for the implementation of synchronized block.

```
class Table
{
    synchronized void printTable(int n)
    {
        for(int i=1;i<=5;i++)
        {
            System.out.println(n*i);
        }
        try
        {
            Thread.sleep(400);
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
    }
}

class MyThread1 extends Thread
{
    Table t;
    MyThread1(Table t)
    {
        this.t=t;
    }
    public void run()
    {
        t.printTable(5);
    }
}

class MyThread2 extends Thread
{
    Table t;
    MyThread2(Table t)
    {
        this.t=t;
    }
    public void run()
    {
        t.printTable(10);
    }
}

public class SynchEx
```

```

{
    public static void main(String[] args)
    {
        Table obj=new Table();
        MyThread1 t1=new MyThread1(obj);
        MyThread2 t2=new MyThread2(obj);
        t1.start();
        t2.start();
    }
}

```

Output

PS C:\Users\OneDrive\Documents\cap615\programs>javac SynchEx.java

PS C:\Users\OneDrive\Documents\cap615\programs> java SynchEx

5
10
15
20
25
10
20
30
40
50

7.2 Exception Handling

An exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime. Exception Handling in Java is one of the powerful mechanisms to handle the runtime errors so that the normal flow of the application can be maintained.

The problem that comes up during the execution of a program is denoted as an exception. This problem can arise due to several reasons, which includes:

1. Supply of invalid data by the user
2. Unavailability of the file required to be accessed
3. Loss of network connection in the middle of communications
4. Shortage of memory in the JVM

Some exceptions occur due to the error generated by the user and the programmer, whereas some other exceptions occur due to the failure of physical resources. An exception can be considered as an object that includes information about the type of error that has occurred.

Whenever any problem occurs while accessing a disk file, Java generates an exception in the form of an object specifying the details of the problem. Thus, whenever a network-related problem is identified, Java wraps up the details of the problem into an object in the form of an exception.

Whenever an error occurs, an object that represents the exception is created and is thrown to the method in which the error occurred. The exception that occurred has to be handled in the same method or can be passed on further in the application, but it has to be caught in the application at some point and processed so that the application does not give any error during run time. The exception object can be generated by the run time system or can be created manually. When an

exception object is created and handed over to the run time system, it is called throwing an exception.



Did you know?

Java mainly deals with uncaught exceptions according to the thread in which they arise.

Apart from generating exceptions for the identification of errors that are external to a program, Java also generates exceptions for irregular code within the program. There are mainly two types of exceptions

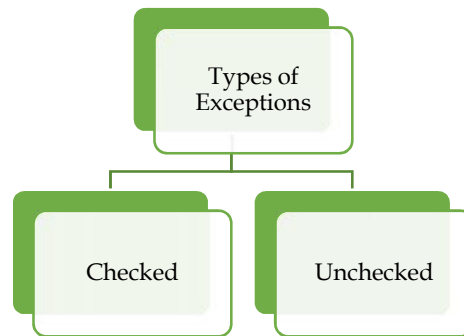


Figure 2: Types of Exceptions

- **Checked:** The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

Examples of checked exceptions are:

1. NoSuchElementException: This exception is thrown when the user tries to use any field or variable in a class that does not exist.
2. ClassNotFoundException: This exception is thrown when the user tries to access a class that is not defined in the program.
3. IllegalAccessException: This exception is thrown when access to a class is denied.
4. InterruptedException: This exception is thrown when a thread is interrupted in a processing, waiting, or sleeping state.
5. NoSuchMethodException: This exception is thrown when the user tries to access a method that does not exist in the program.

- **Unchecked:** The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

Examples of unchecked exceptions are:

1. StringIndexOutOfBoundsException: This exception is thrown when a program tries to access a character at an index of the string that does not exist.
2. ArrayIndexOutOfBoundsException: This exception is thrown when the user tries to access an array index that does not exist.
3. ArithmeticException: This exception is thrown when the user tries to divide a number by zero.
4. NullPointerException: This exception is thrown when the Java Virtual Machine (JVM) tries to execute some operation on an object which points to null data or no data.

Advantage of Exception Handling

The core advantage of exception handling is to maintain the normal flow of the application. An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions.



Example: Suppose there are 10 statements in a Java program and an exception occurs at statement 5; the rest of the code will not be executed, i.e., statements 6 to 10 will not be executed. However, when we perform exception handling, the rest of the statements will be executed. That is why we use exception handling.



Example: Java produces an exception whenever an invalid array of an index is accessed or an illegal class-cast is attempted.

Exception Keywords

try: The "try" keyword is used to indicate a block where an exception code should be placed. This means we can't just utilize the try block. Either catch or finally must come after the try block.

catch: To handle the exception, the "catch" block is invoked. It must be preceded by a try block, thus we can't just use catch block. It can be followed by a later finally block.

throw: The "throw" keyword is used to throw an exception.

throws: Exceptions are declared with the "throws" keyword. It specifies that an exception may occur in the method. There isn't any exception thrown. It's usually used in conjunction with a method signature.

finally: The "finally" block is used to run the program's required code. Whether or whether an exception is handled, it is run.



Task: Write a program to illustrate the use of try and catch and finally keywords.

When only try and catch are used in a program, their combination is called the try-catch block. This block is used to catch a Java exception. Only one exception type can be handled by every catch block, and more than one catch clause can be used in a single try block. In the try-catch block, the try block surrounds a statement that may cause the occurrence of the exception, and the catch block follows the try block. On the occurrence of the exception, a code (that should be executed) is specified in this catch block.

Some of the terminologies used within exception handling are:

Throwing: A process through which an exception is generated and thrown into the program.

Catching: Capturing a currently occurred exception and also executing statements that may help in resolving those exceptions.

Catch Block: The block of code that tries to handle the exception.

Stack Trace: A series of method calls that brings back the control to the point where the exception had occurred.

An exception handler comprises two core sections, namely:

The Try Block: It includes some code, which might throw an exception (generate an error).

The Catch Block: It includes the error handling code. It means that it determines the strategy to be implemented when an error is detected. Thus, exception handling provides a method to separate error handling from the code that may result in errors. It is beneficial in several cases, as it produces clean executable code.



Note: Factors to be Considered for Exception Handling

1. Exception handling does not reduce the amount of work required while handling errors of small programs.
2. The main advantages of exception handling can be observed in a program of considerable size.

7.3 Exception Handling During Multithreading

Exceptions are handled in a single thread by throwing them up or using a try-catch block. The main thread in a multithreaded environment cannot handle the error thrown by the child thread. During multithreading, the following strategies are utilized to handle exceptions:

- In the child thread, try and catch manually (not recommended)
- The thread class includes an `UncaughtExceptionHandler` exception handler. This exception handler can be customized to fit a variety of dimensions.



Caution: The exception class defined by the user must be a subclass of the `Exception` class.



Example

//Implementation of `UncaughtExceptionHandler` strategy

```
class UCEH implements Thread.UncaughtExceptionHandler {
    public void uncaughtException(Thread t, Throwable e)
    {
        System.out.println("And the exception is.. " + e);
    }
}

public class ThreadExcExampleN {
    public static void main(String[] args)
        throws Exception
    {

        Thread.setDefaultUncaughtExceptionHandler(new UCEH());
        throw new Exception("\n language exception is caught");
    }
}
```

Output

```
PS C:\Users\ OneDrive\Documents\cap615>javac ThreadExcExampleN.java
```

```
PS C:\Users\ OneDrive\Documents\cap615> java ThreadExcExampleN
```

```
And the exception is..java.lang.Exception:
```

```
language exception is caught
```

Summary

- Synchronization is the ability to manage several threads' access to a common resource.
- In Java, each object has its monitor, which a thread can lock or unlock. On a display, only one thread can retain a lock at a time.
- An exception is an object that includes information about the type of error that occurred.
- Exception handling is based on the concept of universal error processing, wherein the error correction code is separated from the main body of code and is fed to several exception handlers.
- A checked exception is thrown whenever an error is probable in input-output processing.
- The unchecked exception is thrown due to the invalid argument supplied to a method. These exceptions originate at run-time.

- Keywords such as try, catch, and finally are used for implementing user-defined exceptions.
- For creating an exception, just define a subclass of Exception, which is a subclass of Throwable.

Keywords

Exception: An exception is an occurrence that causes the program's normal flow to be disrupted. It's a type of object that's thrown at runtime.

throws: Exceptions are declared with the "throws" keyword. It specifies that an exception may occur in the method.

Stack Trace: A series of method calls that brings back the control to the point where the exception had occurred.

Synchronization: Synchronisation refers to the ability to regulate multiple threads' access to a shared resource. It's required for thread-to-thread communication to be secure.

Synchronized Block: A synchronized block in Java indicates that a function or a section of code is synchronized. It can only be executed by one thread at a time.

ArithmeticException: This exception is thrown when the user tries to divide a number by zero.

Error: An Error is a Throwable subclass that signals major problems that a reasonable application should not attempt to catch.

Self Assessment

1. Thread synchronization in a process will be required when
 - A. All threads sharing the same address space
 - B. All threads sharing the same global variables
 - C. All threads sharing the same files
 - D. All
2. Which of these keywords are used to implement synchronization?
 - A. synchronize
 - B. syn
 - C. synchronized
 - D. synch
3. What is synchronization about a thread?
 - A. It's a process of handling situations when two or more threads need access to a shared resource
 - B. It's a process by which many threads can access the same shared resource simultaneously
 - C. It's a process by which a method can access many different threads simultaneously
 - D. It's a method that allows too many threads to access any information they require
4. The synchronized keyword is applicable for whom?
 - A. Variables and Methods
 - B. Methods and Classes
 - C. Variables and Classes

D. Methods and Blocks

5. What is the valid syntax for synchronized blocks to get the lock of the current object?

- A. synchronized(this)
- B. synchronized(super)
- C. synchronized(Test.java)
- D. None of these

6. If two threads are trying to execute one synchronized method with two different objects, then their execution order is?

- A. One by One
- B. Both at time
- C. The thread which gets locked first will execute first
- D. None of these

7. Which statement is true?

- A. A static method cannot be synchronized.
- B. If a class has synchronized code, multiple threads can still access the non-synchronized code.
- C. Variables can be protected from concurrent access problems by marking them with the synchronized keyword.
- D. When a thread sleeps, it releases its locks.

8. Assume the following method is properly synchronized and called from a thread A on an object B:

```
wait(2000);
```

After calling this method, when will thread A become a candidate to get another turn at the CPU?

- A. After thread A is notified, or after two seconds.
- B. After the lock on B is released, or after two seconds.
- C. Two seconds after thread A is notified.
- D. Two seconds after lock B is released.

9. Which of the following statements about synchronization are true?

- A. Synchronization is used to coordinate access to objects that are shared among multiple threads.
- B. A synchronized method may be invoked and executed without acquiring an object's lock.
- C. A non-synchronized method requires that a thread obtain an object's lock before the method may be executed.
- D. Another word for a lock is monitor.

10. Which of these keywords is not a part of exception handling?

- A. try
- B. finally

- C. thrown
- D. catch

11. Which of these keywords must be used to handle the exception thrown by try block in some rational manner?

- A. try
- B. finally
- C. throw
- D. catch

12. Which of these keywords is used to manually throw an exception?

- A. try
- B. finally
- C. throw
- D. catch

13. The methods wait(), notify() and notifyAll() must be called only when they own the monitor of that object, otherwise they throw _____.

- A. IllegalMonitorException
- B. IllegalStateException
- C. IllegalMonitorStateException
- D. IllegalMonitorOwnStateException

14. Which of the following correctly represents the exception?

- A. Compile error
- B. Syntax error
- C. Runtime error
- D. Not an error

15. Throwable class has two sub-classes, _____ and _____.

- A. Errors and Exceptions
- B. Error and Exceptions
- C. Errors and Exception
- D. Error and Exception

Answers for Self Assessment

- | | | | | |
|-------|-------|-------|-------|-------|
| 1. D | 2. C | 3. A | 4. D | 5. A |
| 6. B | 7. B | 8. A | 9. D | 10. C |
| 11. D | 12. C | 13. C | 14. C | 15. D |

Review Questions

1. What do you understand by thread synchronization and why it is required? Give the syntax of thread synchronization.
2. Explain the different ways used for the implementation of thread synchronization.
3. With the help of a suitable example explain the different types of exceptions?
4. Write a program to implement the concept of thread synchronization.
5. Elucidate the various strategies used for handling exceptions in a multithreaded environment.
6. Define the following:
 - a. synchronized block
 - b. finally
 - c. Exception
 - d. synchronization



Further Readings

Eckel, B. (2003). *Thinking in JAVA*. Prentice-Hall Professional.

Horstmann, C. S. (1997). *Practical object-oriented development in C++ and Java*. John Wiley & Sons, Inc.

Savitch, W. (2019). *Java: An introduction to problem solving & programming*. Pearson Education Limited.

Oaks, S., & Wong, H. (2004). *Java Threads: Understanding and Mastering Concurrent Programming*. "O'Reilly Media, Inc."

Hyde, P. (1999). *Java thread programming* (p. 528). Sams Pub.



Web Links

<https://www.javatpoint.com/synchronization-in-java>

<https://dotnettutorials.net/lesson/thread-synchronization-in-java/>

https://www.tutorialspoint.com/java/java_thread_synchronization.htm

<https://programmer.group/java-concurrent-multithreading-exception-handling.html>

<http://mapoly.co.in/wpcontent/uploads/2020/03/Chapter4MultithreadingandExceptionHandling.pdf>

Unit 08: Swings

CONTENTS

Objectives

Introduction

8.1 JButton class

8.2 JRadioButton Class

8.3 JTextArea Class

8.4 JComboBox Class

8.5 Constructors ofJComboBox

8.6 JTable Class

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

Objectives

- Learn the basic concept of JButton class and its various Constructors.
- Understand the working of JRadioButton class and its various implementation methods.
- Learn the basic concept of JTextArea Class and its various constructors
- Understand the various methods of JTextArea Class and their Implementation
- Know the basic concept of JComboBox class, its various constructors, methods, and their implementation
- Analyze the working of Jtable class and its various functions

Introduction

Swing is an important component of the Java language, which is provided in the **javax.swing** package. The concept of swings came into existence to provide a more advanced collection of GUI (Graphical User Interface) components to Java as compared to the AWT.



Did you know?

The swing library was introduced by Sun Microsystems to create a GUI, which is elegant, object-oriented, and easy to work with. Swing library is contained in `javax.swing` and `java.awt` packages.

A Swing is a graphical user interface library of Java. It provides multiple platform-independent API interfaces. These interfaces are used for interacting between the users and GUI components. Swing is a part of Java Foundation Class (JFC) and it includes graphical widgets like checkboxes, radio buttons, menus, and so on. The Java Swing can handle all the AWT flexible components. Swing and its components are commonly used in Java because windows' appearance can be changed easily using an important feature of Swings, that is, pluggable look and feel.

8.1 JButton class

The JButton class is used to create a labeled button that has a platform-independent implementation. The application results in some action when the button is pushed. The buttons are the classes derived from the AbstractButton class. The AbstractButton class is a class, which contains different methods that control the behavior of buttons, checkboxes, and radio buttons. Buttons are simple buttons, on which a user can click and perform the desired operations.

JButtons are the swing components that extend the JComponent. The button class in Swings is similar to the button class found in java.awt.Button package. The buttons can be arranged and organized by Actions. Swing buttons are used to display both the text and an image. The letter that is underlined in the text of each button is the keyboard alternative for each button. The appearance of the button is auto-generated with the button's disabled appearance. The user can provide an image as a substitute for the normal image.



Note: A user must implement an action listener while creating buttons.

Declaration JButton class

public class JButton extends AbstractButton implements Accessible

Commonly used Constructors

The following are the types of constructors commonly used in JButton Class:

1. JButton() This constructor will create a button in the component with no icon or text.
2. JButton(String text) This constructor will create a button in the component with the specified text in the parameters.
3. JButton(Icon img) This constructor will create a button in the component with the image img given in the parameters.
4. JButton(String text, Icon img) This constructor will create a button in the component with the text and specified image in the parameters.



Example :// Implementation of JButton() constructor.

```
import java.awt.*;
import javax.swing.*;

public class JButton1 {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JButton First Example");
        JButton b = new JButton();
        b.setPreferredSize(new Dimension(100, 30));
        frame.add(b);
        frame.setSize(500, 500);
        frame.setLayout(new FlowLayout());
        frame.setVisible(true);
    }
}
```

Output

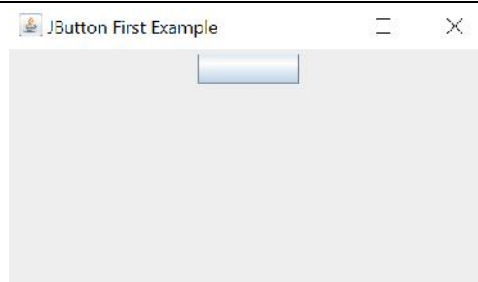


Figure 1: Output of JButton



Task :Write a simple Java program to illustrate the concept of JButton



Example: //Implementation of JButton(String Text)

```
import java.awt.*;
import javax.swing.*;

public class JButton2 {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JButton Example with Text");
        JButton b = new JButton("Cancel");
        b.setPreferredSize(new Dimension(100, 30));
        frame.add(b);
        frame.setSize(500, 500);
        frame.setLayout(new FlowLayout());
        frame.setVisible(true);
    }
}
```

Output :

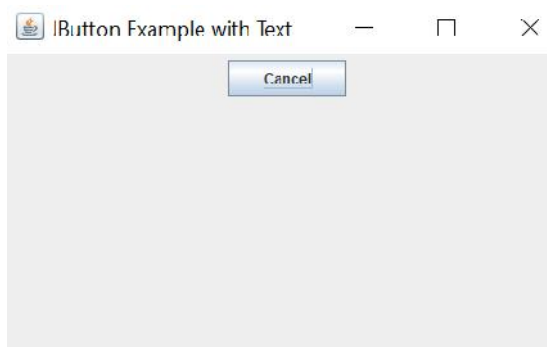


Figure 2: Output of JButton(String Text)



Example: //Implementation of JButton(img icon) constructor.

```
import java.awt.*;
import javax.swing.*;

public class JButton3 {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JButton Example for image");
    }
}
```

Programming in JAVA

```

JButton b=new JButton(new ImageIcon("C:\\Users\\Harjinder Kaur\\Downloads\\cc.jpg"));
frame.add(b);
frame.setSize(100,100);
frame.setLayout(new FlowLayout());
frame.setVisible(true);
}
}

```

Output

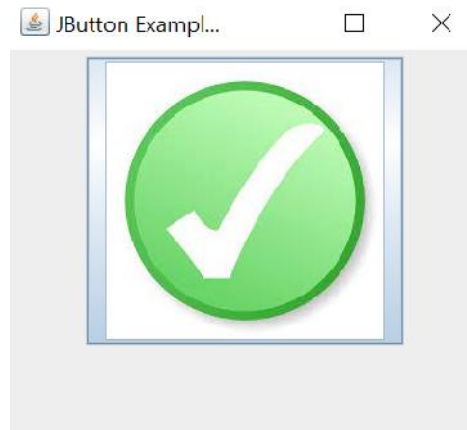


Figure 3: Output of JButton(img icon) constructor.



Example: // Implementation of JButton(String,Img) constructor.

```

import java.awt.*;
import javax.swing.*;

public class JButton4 {
    public static void main(String[] args) {
        JFrame frame=new JFrame("JButton Example for Image and Text");
        JButton b=new JButton("submit",new ImageIcon("C:\\Users\\Harjinder
        Kaur\\Downloads\\images.jfif"));
        frame.add(b);
        frame.setSize(500,500);
        frame.setLayout(new FlowLayout());
        frame.setVisible(true);
    }
}

```

Output :

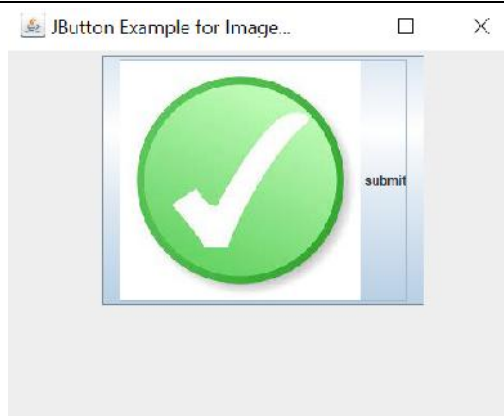


Figure 4: Output of JButton(String,Img) constructor.

8.2 JRadioButton Class

A Radio button is also a Swing component, which is defined as an item that a user can select or deselect with a single click. From a group of radio buttons, only one can be selected per click, unlike checkboxes. Before the creation of any radio buttons, the user must create an instance of that button, the `ButtonGroup` object. Then, the user must add the radio buttons in the `ButtonGroup` object. The radio button is used to select one option from multiple options.

As a Swing component, a radio button is written as **JRadioButton**. A `JRadioButton` object can be created using many constructors. Some of these constructors are:

- **JRadioButton()** : Creates a unselected `RadioButton` with no text.
- **JButton(String s)** Creates a `JButton` with a specific

Methods Used in JRadioButton

ButtonGroup(): It is used to create a group, in which we can add `JRadioButton`. We can select only one `JRadioButton` in a `ButtonGroup`.

Steps to Group the radio buttons together.

- ✓ Create a `ButtonGroup` instance by using the "`ButtonGroup()`" Method.
- ✓ `ButtonGroup G = new ButtonGroup()`
- ✓ Now add buttons in a Group "`G`", with the help of "`add()`" Method.



Example: `G.add(Button1);G.add(Button2);`

isSelected() : it will return a Boolean value true or false, if a `JRadioButton` is selected it will return true otherwise false.



Example: `JRadioButton.isSelected()`

Set(...) and Get(...) Methods: Set and get are used to replace directly accessing member variables from external classes. Instead of accessing class member variables directly, you define get methods to access these variables, and set methods to modify them.

The `RadioButton` class can be implemented by one of the following:

Using ActionListener



Example: //Implementation of `JRadioButtonWithActionListener`

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
class Demo extends JFrame {
```

```
// Declaration of object of JRadioButton class.
JRadioButton jRadioButton1;
// Declaration of object of JRadioButton class.
JRadioButton jRadioButton2;
// Declaration of object of JButton class.
JButton jButton;
// Declaration of object of ButtonGroup class.
ButtonGroup G1;
// Declaration of object of JLabel class.
JLabel L1;
// Constructor of Demo class.
public Demo()
{
    // Setting layout as null of JFrame.
    this.setLayout(null);
    // Initialization of object of "JRadioButton" class.
    jRadioButton1 = new JRadioButton();
    // Initialization of object of "JRadioButton" class.
    jRadioButton2 = new JRadioButton();
    // Initialization of object of "JButton" class.
    jButton = new JButton("Click");
    // Initialization of object of "ButtonGroup" class.
    G1 = new ButtonGroup();
    // Initialization of object of "JLabel" class.
    L1 = new JLabel("Qualification");
    // setText(...) function is used to set text of radio button.
    // Setting text of "jRadioButton2".
    jRadioButton1.setText("Under-Graduate");
    // Setting text of "jRadioButton4".
    jRadioButton2.setText("Graduate");
    // Setting Bounds of "jRadioButton2".
    jRadioButton1.setBounds(120, 30, 120, 50);
    // Setting Bounds of "jRadioButton4".
    jRadioButton2.setBounds(250, 30, 80, 50);
    // Setting Bounds of "jButton".
    jButton.setBounds(125, 90, 80, 30);
    // Setting Bounds of JLabel "L2".
    L1.setBounds(20, 30, 150, 50);
    // "this" keyword in java refers to current object.
    // Adding "jRadioButton2" on JFrame.
    this.add(jRadioButton1);
```

```

        // Adding "jRadioButton4" on JFrame.
        this.add(jRadioButton2);
        // Adding "jButton" on JFrame.
        this.add(jButton);
        // Adding JLabel "L2" on JFrame.
        this.add(L1);
// Adding "jRadioButton1" and "jRadioButton3" in a Button Group "G2".
        G1.add(jRadioButton1);
        G1.add(jRadioButton2);
        // Adding Listener to JButton.
        jButton.addActionListener(new ActionListener() {
            // Anonymous class.
            public void actionPerformed(ActionEvent e)
            {
                // Declaration of String class Objects.
                String qual = " ";
                // If condition to check if jRadioButton2 is selected.
                if (jRadioButton1.isSelected()) {
                    qual = "Pass";
                }
                else if (jRadioButton2.isSelected()) {
                    qual = "Fail";
                }
                else {
                    qual = "NO Button selected";
                }
            }
        });
// MessageDialog to show information selected radion buttons.
        JOptionPane.showMessageDialog(Demo.this, qual);
    }
}

class RadioButton {
    // Driver code.
    public static void main(String args[])
    { // Creating object of demo class.
        Demo f = new Demo();
        // Setting Bounds of JFrame.
        f.setBounds(100, 100, 400, 200);
        // Setting Title of frame.
        f.setTitle("RadioButtons");
    }
}

```

Programming in JAVA

```

        // Setting Visible status of frame as true.
        f.setVisible(true);
    }
}

```

Output:

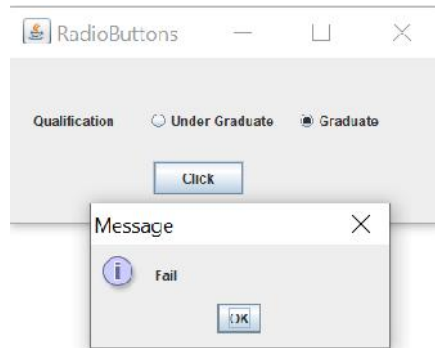


Figure 5: Output of JRadioButtonWithActionListener

Without using ActionListener.



Example://Implementation of JRadioButton Without ActionListener

```

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

class Demo extends JFrame {
    // Declaration of object of JRadioButton class.
    JRadioButton jRadioButton1;
    // Declaration of object of JRadioButton class.
    JRadioButton jRadioButton2;
    // Declaration of object of JButton class.
    JButton jButton;
    // Declaration of object of ButtonGroup class.
    ButtonGroup G1;
    // Declaration of object of JLabel class.
    JLabel L1;

    // Constructor of Demo class.
    public Demo()
    {
        // Setting layout as null of JFrame.
        this.setLayout(null);
        // Initialization of object of "JRadioButton" class.
        jRadioButton1 = new JRadioButton();
        // Initialization of object of "JRadioButton" class.

```

```

jRadioButton2 = new JRadioButton();
// Initialization of object of "JButton" class.
jButton = new JButton("Click");
// Initialization of object of "ButtonGroup" class.
G1 = new ButtonGroup();
// Initialization of object of "JLabel" class.
L1 = new JLabel("Qualification");
// setText(...) function is used to set text of radio button.
// Setting text of "jRadioButton2".
jRadioButton1.setText("Under-Graduate");
// Setting text of "jRadioButton4".
jRadioButton2.setText("Graduate");
// Setting Bounds of "jRadioButton2".
jRadioButton1.setBounds(120, 30, 120, 50);
// Setting Bounds of "jRadioButton4".
jRadioButton2.setBounds(250, 30, 80, 50);
// Setting Bounds of "jButton".
jButton.setBounds(125, 90, 80, 30);
// Setting Bounds of JLabel "L2".
L1.setBounds(20, 30, 150, 50);
// "this" keyword in java refers to current object.
// Adding "jRadioButton2" on JFrame.
this.add(jRadioButton1);
// Adding "jRadioButton4" on JFrame.
this.add(jRadioButton2);
// Adding "jButton" on JFrame.
this.add(jButton);
// Adding JLabel "L2" on JFrame.
this.add(L1);
// Adding "jRadioButton1" and "jRadioButton3" in a Button Group "G2".
G1.add(jRadioButton1);
G1.add(jRadioButton2);
}
}
class RadioButton {
// Driver code.
public static void main(String args[])
{ // Creating object of demo class.
    Demo f = new Demo();
// Setting Bounds of JFrame.
f.setBounds(100, 100, 400, 200);

```

```
// Setting Title of frame.
f.setTitle("RadioButtons");

// Setting Visible status of frame as true.
f.setVisible(true);

}

}
```

Output:



Figure 6: Output of JRadioButtonWithoutActionListener

8.3 JTextArea Class

The **JTextArea** component is used to display plaintext and is a multi-line text area. This is a swing text component. It is known as a lightweight component for working with text. This text component does not handle scrolling, and for this reason, the **JScrollPane** component is used. **JTextArea** inherits **JComponent** class.

The text in **JTextArea** can be set to different available fonts and can be appended to new text. A text area can be customized to the need of the user.

Constructors of JTextArea

The following constructors are used to create the **JTextArea** class;

1. **JTextArea()** : constructs a new blank text area **JTextArea(String s)**: constructs a new text area with a given initial text.
2. **JTextArea(int row, int column)**: constructs a new text area with a given number of rows and columns.
3. **JTextArea(String s, int row, int column)**: constructs a new text area with a given number of rows and columns and a given initial text.

JTextArea Class Declaration

```
public class JTextArea extends JTextComponent
```

Methods of JTextArea Class

The following are the various methods used with **JTextArea** Class:

- **append(String s)**: appends the given string to the text of the text area.
- **getLineCount()** : get number of lines in the text of text area.
- **setFont(Font f)**: sets the font of the text area to the given font.
- **setColumns(int c)**: sets the number of columns of the text area to a given integer.
- **setRows(int r)**: sets the number of rows of the text area to a given integer.
- **getColumns()** :get the number of columns of text area.
- **getRows()**: get the number of rows of the text area.



Example ://Implementation of JTextArea Class

```
import javax.swing.*;

public class TextAreaExample
```

```
{
TextAreaExample(){
JFrame f= new JFrame();
JTextArea area=new JTextArea();
area.setText("This is a string of text.\n");
area.setBounds(10,30, 200,200);
area.append("Welcome to this world");
f.add(area);
f.setSize(300,300);
f.setLayout(null);
f.setVisible(true);
String str = area.getText();
}
public static void main(String args[])
{
newTextAreaExample();
}
}
```

Output:

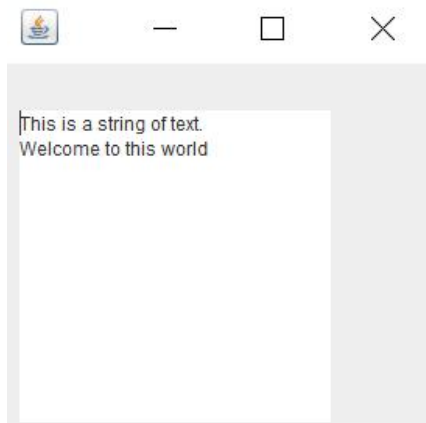


Figure 7: Output of JTextArea

8.4 JComboBox Class

JComboBox is a part of the Java Swing package. JComboBox inherits JComponent class. It shows a popup menu that shows a list and the user can select an option from that specified list. JComboBox can be editable or read-only depending on the choice of the programmer. A Swing component, which brings a button or editable field and a drop-down list together, from which the user can select any value, is referred to as a **ComboBox**. The **ComboBox** can be made editable also. For doing this, an editable field is added into the **ComboBox**, in which, the user can type a value.

A **ComboBox** contains three components, namely, a text field, a button, and a list. These components together perform some functions. They help in:

Programming in JAVA

- (a) Displaying the selected text field on the screen.
- (b) Exhibiting the display of the list box that is controlled by the button that is present on the rightside of the text field.
- (c) Editing the selected text field.
- (d) Displaying icons along with or in the place of the text.

8.5 Constructors of JComboBox

The following table shows the various constructors used to create Combobox along with their description :

Table 1: JComboBox Constructors

Constructor	Description
JComboBox()	Creates a JComboBox with a default data model.
JComboBox(Object[] items)	Creates a JComboBox that contains the elements in the specified array.
JComboBox(Vector<?> items)	Creates a JComboBox that contains the elements in the specified Vector.

Methods of JComboBox

Every class is having its own methods which are used for the implementation. The below is the list of various methods used in JComboBox Class:

- **void addItem(Object anObject):** It is used to add an item to the item list.
- **void removeItem(Object anObject) :** It is used to delete an item to the item list.
- **void removeAllItems():** It is used to remove all the items from the list.
- **void setEditable(boolean b):** It is used to determine whether the JComboBox is editable.
- **void addActionListener(ActionListener a):** It is used to add the ActionListener. The Java ActionListener is notified whenever you click on the button or menu item. It is notified against ActionEvent. The ActionListener interface is found in java.awt.event package. It has only one method: actionPerformed().
- **void addItemListener(ItemListener i) :** The Java ItemListener is notified whenever you click on the checkbox. It is notified against ItemEvent. The ItemListener interface is found in java.awt.event package. It has only one method: itemStateChanged().



Example: // Implementation of JComboBox class

```
import javax.swing.*;

public class ComboExample {
    JFrame f;

    ComboExample(){
        f=new JFrame("Implementation of JComboBox");
        String subject[]={"English","Hindi","Science","Database","C++"};
        JComboBox cb=new JComboBox(subject);
        cb.setBounds(50, 50,90,20);
        f.add(cb);
    }
}
```

```

f.setLayout(null);
f.setSize(400,500);
f.setVisible(true);
}
public static void main(String[] args) {
newComboExample();
}
}

```

Output:

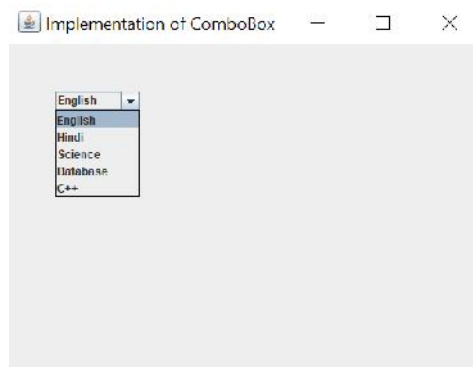


Figure 8: Output of JComboBox



Task: Compile a list of constructors used in the JTextArea and JComboBox class, along with their description.

8.6 JTable Class

The JTable class is used to display data in tabular form. It is composed of rows and columns.

Constructors used in Jtable

- JTable(): Creates a table with empty cells.
- JTable(int rows, int cols): Creates a table of size rows * cols.
- JTable(Object[][] data, Object []Column): Creates a table with the specified data.

The following are the various functions used in JTable class:

- **addColumn(TableColumn []column):** This method adds a column at the end of the JTable.
- **clearSelection():** It deSelects all the selected rows and columns.
- **editCellAt(int row, int col):** The method edits the intersecting cell of the column number col and row number row programmatically, if the given indices are valid and the corresponding cell is editable.
- **setValueAt(Object value, int row, int col):** It sets the cell value as 'value' for the position row, col in the JTable.



Example: //Implementation of Jtable Class

```

import javax.swing.*;
public class jtExample {
JFrame f;
jtExample(){

```

Programming in JAVA

```

f=new JFrame();

String tab[][]={ {"1","Rahul","6"},
                 {"2","Kawar","7"},
                 {"3","Seerat","8"};

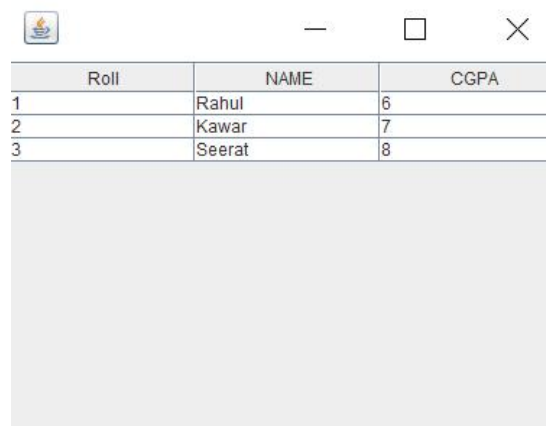
String col[]={"Roll","NAME","CGPA"};

JTablejt=new JTable(tab,col);
jt.setBounds(30,40,200,300);
JScrollPane sp=new JScrollPane(jt);
f.add(sp);
f.setSize(300,400);
f.setVisible(true);
}

public static void main(String[] args) {
newJtExample();
}
}

```

Output :



Roll	NAME	CGPA
1	Rahul	6
2	Kawar	7
3	Seerat	8

Figure 9: Output of JTable

Summary

- Swing is a part of Java Foundation Class (JFC) and it includes graphical widgets like checkboxes, radio buttons, menus, and so on.
- The AbstractButton class is a class, which contains different methods that control the behavior of buttons, checkboxes, and radio buttons.
- A radio button is a Swing component, which is similar to the checkbox component. But in a group of radio buttons, only one radio button can be selected per click, unlike checkboxes.
- A Radio button is also a Swing component, which is defined as an item that a user can select or deselect with a single click.

- AWT package **javax.swing** contains all the Swing components; namely: JComponent, JButton, JCheckBox, JFileChooser, JTextField, JFrame, JPanel, JApplet, JOptionPane, JDialog, and JColorChooser.
- The concept of Swings in Java uses AWT's component. The architecture of the Swing components modifies the program's appearance and behavior.
- A **ComboBox** contains three components, namely, a text field, a button, and a list.
- The **JTextArea** component is used to display plain text and is a multi-line text area.

Keywords

java.awt.event: A package in Java that provides interfaces and classes, which are used to deal with various events that are fired by the AWT components.

javax.swing: A package in Java that provides a collection of lightweight components, that is, the components working similarly on all platforms.

JTable: The JTable class is used to display data in tabular form.

JTextArea: The JTextArea component is used to display plain text and is a multi-line text area. This is a swing text component.

ButtonGroup: ButtonGroup object is used to create a group of the button in which only one button at a time can be selected.

Self Assessment

1. Which of the following components extends and matches the Frame class in the AWT package?
A. JFrame
B. JPanel
C. JApplet
D. JDialog
2. JCheckBox and JCheckBoxMenuItem are inherited from which component's class?
A. ComboBoxes
B. Lists
C. Menu
D. Button
3. Which component is swing represents data in rows and columns?
A. JTextArea
B. JTable
C. JPanel
D. JtabbedPane
4. Which of the following is the correct declaration of JTextArea Class?
A. JTextComponent extends JTextArea
B. JTextArea extends JTextComponent
C. JTextField extends JTextArea

D. JTextArea extends JTextField

5. Which of the following objects are created by using the JRadioButton() and JRadioButton(Icon icon) constructors?

- A. JRadioButton
- B. JButton
- C. JTextBox
- D. JCheckBox

6. JFrame, JPanel, JLabel, JButton are examples of _____

- A. methods
- B. classes
- C. interfaces
- D. packages

7. What code adds this JButton to this JPanel?

```
JPanel panel = new JPanel();
```

```
JButton button = new JButton();
```

- A. panel.add(button);
- B. button.add(panel);
- C. frame.add(panel);
- D. panel.add(JButton);

8. Which of these events is generated when a button is pressed?

- A. KeyEvent
- B. WindowEvent
- C. AdjustmentEvent
- D. ActionEvent

9. JButtons generate what type of event?

- A. ActionEvents
- B. ChangeEvents
- C. WindowEvents
- D. MouseEvents

10. Which of the following packages is used to include classes to create the user interface like Button and Checkbox?

- A. java.lang
- B. java.net
- C. java.awt
- D. java.io

11. Which of the following creates an initially unselected radio button with no set text.

- A. JRadioButton(Icon icon)
- B. JRadioButton()
- C. JRadioButton(Action a)
- D. All

12. In swing, Buttons are subclasses of which class?

- A. AbstractButton
- B. JButton
- C. Button
- D. JRadioButton

13. Which component provides support for two- state buttons?

- A. Button, Checkbox
- B. JCheckBox, JRadioButton
- C. Button, MenuItem
- D. JCheckBox, TextField

14. What is the purpose of Jtable?

- A. JTable object displays data in Table form with header detail.
- B. JTable object displays rows and columns of data.
- C. JTable object displays rows of data with a header.
- D. JTable object displays columns of data

15. Which of the following is/are the constructors of JComboBox?

- A. JComboBox()
- B. JComboBox(Object[] items)
- C. JComboBox(Vector<?> items)
- D. All

Answers for Self Assessment

- | | | | | |
|-------|-------|-------|-------|-------|
| 1. A | 2. D | 3. B | 4. B | 5. A |
| 6. B | 7. A | 8. D | 9. A | 10. C |
| 11. B | 12. A | 13. B | 14. B | 15. D |

Review Questions

1. "A JRadioButton object can be created using many constructors." Discuss these constructors.
2. "A JComboBox contains three components." Discuss these components.
3. What is the use of JComboBox class? With an example explain the various constructors of JComboBox class?

Programming in JAVA

4. With an example explain the various constructors and methods of JTextArea class.
5. "JTable class is used to display data in tabular form" justify the statement with the help of a program.
6. Write a program to show the implementation of the following:
 - a) JButton class.
 - b) JComboBox class.

**Further Readings**

Er. R. Kabilan, (2009), Secrets of JAVA, Firewall Media.

Kim Topley, (2000), Core Swing: advanced programming, Prentice Hall PTR.

Arnold, K., Gosling, J., & Holmes, D. (2005). *The Java programming language*. Addison Wesley Professional.

Haggar, P. (2000). *Practical Java: programming language guide*. Addison-Wesley Professional.

**Web Links**

<http://java.comsci.us/examples/swing/JTextArea.html>

<http://www.javabeginner.com/java-swing/java-swing-tutorial>

<http://zetcode.com/tutorials/javaswingtutorial/basicswingcomponentsII>

<http://www.devx.com/tips/Tip/12812>

<http://www.beginner-java-tutorial.com/jbutton.html>

https://ocw.mit.edu/courses/civil-and-environmental-engineering/1-00-introduction-to-computers-and-engineering-problem-solving-spring-2012/lecture-notes/MIT1_00S12_Lec_17.pdf

Unit 09: More on Swings

CONTENTS

Objectives

Introduction

9.1 JColorChooser Class

9.2 JProgressBar

9.3 JSlider Class

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

Objectives

After this unit you will be able to:

- Learn the basic concept of JColorChooser, JProgressBar, and JSlider Class.
- Understand the various constructors and methods of JColorChooser, JProgressBar, and JSlider Class.
- Implementation of JColorChooser, JProgressBar and JSlider Class.

Introduction

Swing is a Java GUI widget toolkit. It's part of Oracle's Java Foundation Classes (JFC), which provides an API for creating graphical user interfaces for Java programs. It was created to give a more advanced set of graphical user interface components than the previous Abstract Window Toolkit (AWT). Swing supports a pluggable look and feels that allows applications to have a look and feel that is unconnected to the underlying platform, as well as a look and feel that emulates the look and feel of numerous platforms. Its components are more powerful and flexible than those of AWT.

Swing is a framework developed entirely in Java. The 'javax.swing' package in Java contains the Swing framework. The javax.swing package contains classes that begin with the letter 'J.' JProgressBar, JSlider, JColorChooser, and other classes that will be found in the javax.swing package. Every control is described in the javax.swing package that is included in AWT is available in the Swing API. As a result, swing serves as a sort of AWT replacement. Swing also supports several complex components tabbed windows.



Did you Know?

In Java, there are 250 new classes and 75 interfaces in Swing.

9.1 JColorChooser Class

The JColorChooser class is used to create a color selector dialogue box that allows the user to choose any color. It is a descendant of the JComponent class. The JColorChooser class is a javax.swing package pre-defined class that displays a color dialogue box. This JColorChooserdialogue box allows you to choose a color for a specific reason. JColorChooser is a

Programming in JAVA

new Swing component; no corresponding class exists in AWT. There are five panes in the default color picker:

Swatches: Swatches are used to select a color from a set of swatches.

HSV: For using the Hue-Saturation-Value color representation to select a color. It was previously known as HSB (Hue-Saturation-Brightness) until JDK 7.

HSL: HSL stands for Hue-Saturation-Lightness and is used to select a color.

RGB: The Red-Green-Blue color model is used to select a color.

CMYK: CMYK is a color model that uses the process color or four color model to select a color.

Constructors of JColorChooser Class

The following are the commonly used constructors of JColorChooser class:

JColorChooser(): Creates a color picker pane with a white background.

JColorChooser(Color initialColor): The specified initial color is used to create a color chooser pane.

JColorChooser(ColorSelectionModel model): Creates a colour picker pane with the ColorSelectionModel supplied.

Methods used in JColorChooser Class

The following table list the various methods of JColorChooser class along with their description:

Table 1: Methods of JColorChoose

JColorChooser	Description
setColor(Color color)	The color chooser's current color is changed to the color supplied.
setColor(int c)	Sets the color chooser's current color to the provided color.
setColor(int r, int g, int b)	Sets the color chooser's current color to the provided RGB color.
showDialog(Component cmp, String title, Color init_Color)	Displays a modal color-selection dialogue and blocks until it is hidden.
setChooserPanels(AbstractColorChooserPanel[] panels)	The Color Panels that were utilised to pick a colour value are specified here.
addChooserPanel(AbstractColorChooserPanel panel)	A color chooser panel is added to the color-chooser.
setSelectionModel(ColorSelectionModel newModel)	Sets the model that contains the chosen colour.
setPreviewPanel(JComponent preview)	Using this method the current preview panel is set.



Example

//Implementation of JColorChooser to change background color.

```
import java.awt.*;
```

```
import java.awt.event.*;
```

```
import javax.swing.*;
```

```
import javax.swing.*;
```

```
class JCCE implements ActionListener
{
    JButton jb;

    void launchFrame()
    {
        JFrame jf = new JFrame("example to change background color");
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jb=new JButton("select background color");
        jb.addActionListener(this);
        jf.add(jb, BorderLayout.CENTER);
        jf.setSize(300, 200);
        jf.setVisible(true);
    }

    public void actionPerformed(ActionEvent actionEvent)
    {
        Color initialbg = jb.getBackground();
        Color bg = JColorChooser.showDialog(null,"selection colors",initialbg);
        if (bg!= null)
        {
            jb.setBackground(bg);
        }
    }

    public static void main(String args[])
    {
        JCCE obj=new JCCE();
        obj.launchFrame();
    }
}
```

Output

PS C:\Users\ OneDrive\Documents\cap615\programs>javac JCCE.java

PS C:\Users\ OneDrive\Documents\cap615\programs> java JCCE

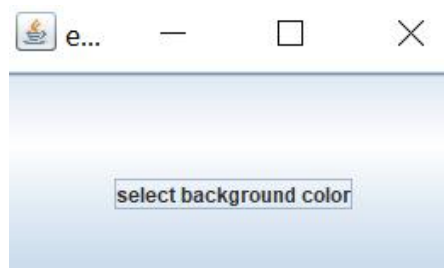


Figure 1: Default Background Color

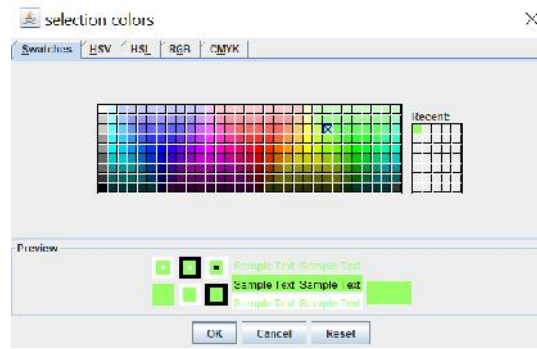


Figure 2: Color Selection

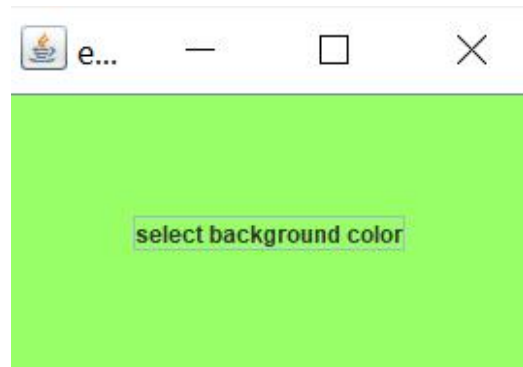


Figure 3: Modified background



Task: Write a simple Java program to illustrate the concept of JColorChooser Class.

9.2 JProgressBar

The JProgressBar is a Java Swing component. JProgressBar is a visual representation of the progress of a task. JProgressBar displays the percentage of a task that has been completed. As the work is completed, the progress bar fills up. It can also display additional text in addition to the percentage of task completion.



Figure 4: Progress Bar

Declaration of JProgressBar class

public class JProgressBar extends JComponent implements SwingConstants, Accessible

Constructors of JProgressBar

JProgressBar(): This constructor generates a progress bar without any text.

JProgressBar(int orientation): It produces an orientation-specific progress bar. If SwingConstants.VERTICAL is being passed then a vertical progress bar is created and if SwingConstants.HORIZONTAL is being passed then a horizontal progress bar is created.

JProgressBar(int min, int max): It generates a progress bar with a minimum and maximum value supplied.

JProgressBar(int orientation, int min, int max): This constructor generates a progress bar with a minimum and maximum value, as well as an orientation.

Methods of JProgressBar class

The following shows indicated the various methods used in the implementation of JProgressBar to get different types of bars:

Unit 09: More on Swings

getMaximum(): It returns the maximum value of the progress bar.

getMinimum(): It returns the minimum value of the progress bar.

String getString(): This method returns the string representation of the current value in the progress bar.

void setMaximum(int n): This method sets the maximum value of the progress bar to n.

void setMinimum(int n): It sets the minimum value of the progress bar to n.

void setValue(int n): This method sets the current value of the Progress bar to n.

void setString(String s): It sets the value of the progress String to the String s using setString(String s).

getMaximum(): It is used to returns the maximum value of the progress bar.

getMinimum(): This method is used to returns the minimum value of the progress bar.

getString(): It returns the string representation of the current value in the progress bar.

void setMaximum(int n): It is used to sets the progress bar's maximum value to n.



Example

// Program to implement JProgressBar

```
import javax.swing.*;

public class PBExample extends JFrame{
    JProgressBar jb;
    int i=0,num=0;
    PBExample(){
        jb=new JProgressBar(SwingConstants.HORIZONTAL);
        jb.setBounds(40,40,160,30);
        jb.setValue(0);
        jb.setStringPainted(true);
        add(jb);
        setSize(250,150);
        setLayout(null);
    }
    public void iterate(){
        while(i<=100){
            jb.setValue(i);
            i=i+10;
            try{ Thread.sleep(150);}catch(Exception e){}
        }
    }
    public static void main(String[] args) {
        PBExample pb=new PBExample();
        pb.setVisible(true);
        pb.iterate();
    }
}
```

Output

Programming in JAVA

PS C:\Users\ OneDrive\Documents\cap615\programs>javac PBExample.java

PS C:\Users\ OneDrive\Documents\cap615\programs> java PBExample

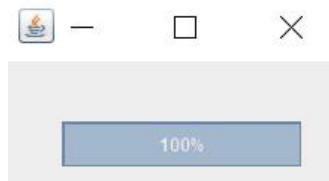


Figure 5: JProgressBar Output



Example

```
// Java Program to set specific string message on progressbar
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class barprogress extends JFrame {
    // create a frame
    static JFramejf;
    static JProgressBarjb;
    public static void main(String args[])
    {
        // create a frame
        jf = new JFrame("ProgressBar with string message");
        // create a panel
        JPaneljp = new JPanel();
        // create a progressbar
        jb = new JProgressBar();
        // set initial value
        jb.setValue(0);
        jb.setStringPainted(true);
        // add progressbar
        jp.add(jb);
        jf.add(jp);
        jf.setSize(500, 500);
        jf.setVisible(true);
    }
    //new progress();
    fill();
}
// function to increase progress
public static void fill()
{
    int i = 0;
```

```

try {
    while (i <= 100) {
        // set text according to the level to which the bar is filled
        if (i > 30 && i < 70)
            jb.setString("wait for sometime");
        else if (i > 70)
            jb.setString("almost finished loading");
        else
            jb.setString("loading started");
        // fill the menu bar
        jb.setValue(i + 10);
        // delay the thread
        Thread.sleep(3000);
        i += 20;
    }
}
catch (Exception e) {
}
}
}

```

Output

PS C:\Users\ OneDrive\Documents\cap615\programs>javac barprogress.java

PS C:\Users\ OneDrive\Documents\cap615\programs> java barprogress

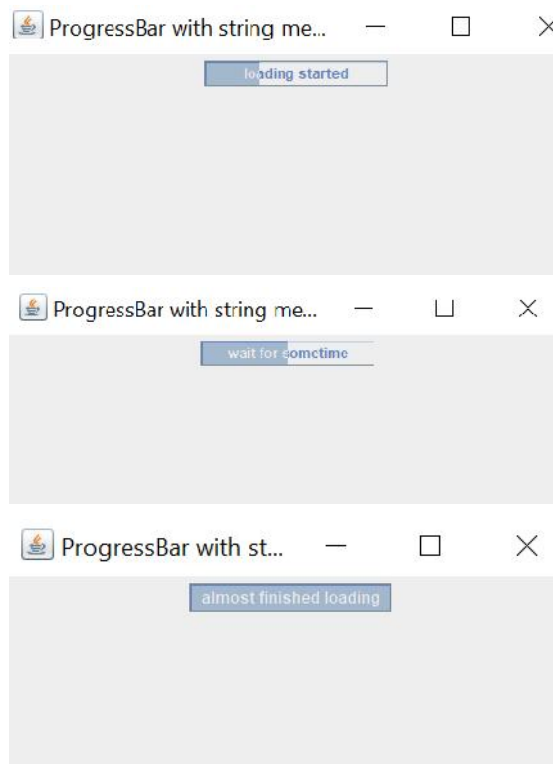


Figure 6: Progressbar at Different Stages

9.3 JSlider Class

A slider is built/ designed by JSlider. A user can select a value from a range using JSlider. Create an instance of the slider class using the JSlider() constructor.

What exactly is a slider?

Because a slider is constrained by a minimum and maximum value, the user must choose from within that range when entering a numeric number. The Java Swing package includes JSlider. JSlider is a slider implementation. The user can choose a value by sliding the knob inside the restricted value.

Between two large tick marks, the slider might show major tick markings as well as minor tick marks. Only those points can be used to position the knob.

The following illustration explains what a slider is:

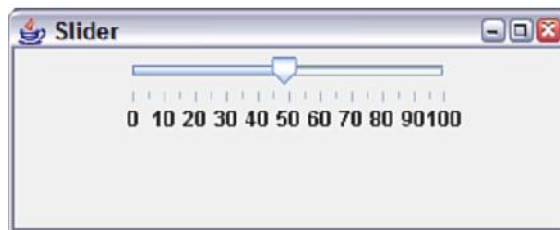


Figure 7: Slider

Constructors of JSlider

The following are some of the most common constructors in JSlider:

JSlider(): It produces a slider with a value of 50 as the initial point and a range of 0 to 100 as the range.

JSlider(BoundedRangeModel b): Using the provided BoundedRangeModel, creates a horizontal slider.

JSlider(int orientation): It constructs a slider with any JSlider's provided orientation. JSlider.HORIZONTAL or JSlider.VERTICAL with a range of 0 to 100 and a starting value of 50.

JSlider(int min, int max): Creates a horizontal slider with an initial value equal to the average of the min and max values, using the provided min and max values.

JSlider(int min, int max, int value): The stated min, max, and value are used to create a horizontal slider.

JSlider(int orientation, int min, int max, int value): Creates a slider with the supplied minimum, maximum, and starting values, as well as the specified orientation.

Methods used with JSlider

void setMinorTickSpacing(int p): Sets the minor tick spacing in the slider.

void setMajorTickSpacing (int p): Sets the major tick spacing.

void setMinimum(int p): sets the slider's minimum value.

void setMaximum(int p): sets the slider's maximum value.

void setPaintsTicks(boolean bl): determines whether or not a tick mark should be painted.

void setPaintLabels(boolean bl): checks if labels have been painted.

void setPaintTracks(boolean bl): Sets whether the track is painted.



Example

```
//Implementation of Jslider class without ticks
```

```
import javax.swing.*;
```

```

public class SliderEx extends JFrame
{
    public SliderEx()
    {
        JSlider slider=new JSlider();
        JPanel panel=new JPanel();
        panel.add(slider);
        add(panel);
    }
    public static void main(String s[])
    {
        SliderEx frame=new SliderEx();
        frame.pack();
        frame.setVisible(true);
    }
}

```

Output

PS C:\Users\ OneDrive\Documents\cap615\programs>javac SliderEx.java

PS C:\Users\ OneDrive\Documents\cap615\programs> java SliderEx



Figure 8: Slider without Ticks

We need to know the positions of specific numbers in some instances. In this case, we can add the positions to the slider and name them.

**Example**

//Implementation of major and minor ticks

```

import javax.swing.*;

public class SliderTicks extends JFrame
{
    public SliderTicks()
    {
        JSlider sl=new JSlider(JSlider.VERTICAL,0,80,35);
        sl.setMinorTickSpacing(2);
        sl.setMajorTickSpacing(10);
        sl.setPaintTicks(false);
        sl.setPaintLabels(true);
    }
}

```

Programming in JAVA

```

JPanel panel=new JPanel();
panel.add(sl);
add(panel);
}
public static void main(String s[])
{
SliderTicksst=new SliderTicks();
st.pack();
st.setVisible(true);
}
}

```

Output

PS C:\Users\ OneDrive\Documents\cap615\programs>javac SliderTicks.java

PS C:\Users\ OneDrive\Documents\cap615\programs> java SliderTicks

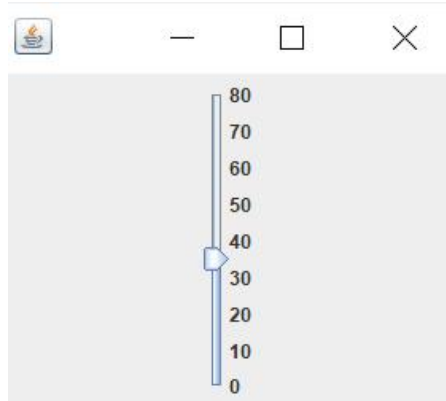


Figure 9: Slider with values

We can add the change listener to the slider and conduct several operations to make it more functional. Another panel might be added to show the value of the slider in real-time, for example. Then we know exactly where the slider is. In the next example, we add a change listener to the slider and display the slider's value in a panel below the slider.

**Example**

// java Program to create a vertical slider with min and max value and major and minor ticks painted and set the font of the slider.

```

import javax.swing.event.*;
import java.awt.*;
import javax.swing.*;

class slfont extends JFrame implements ChangeListener {
    // frame
    static JFramejf;
    // slider
    static JSliderjs;
    // label
    static JLabeljl;
    // main class

```

```
public static void main(String[] args)
{
    // create a new frame
    jf = new JFrame("My frame");
    // create a object
    slfont sf = new slfont();
    // create label
    jl = new JLabel();
    // create a panel
    JPanel jp = new JPanel();
    // create a slider
    js = new JSlider(0, 200, 120);
    // paint the ticks and tracks
    js.setPaintTrack(true);
    js.setPaintTicks(true);
    js.setPaintLabels(true);
    // set spacing
    js.setMajorTickSpacing(50);
    js.setMinorTickSpacing(5);
    // setChangeListener
    js.addChangeListener(sf);
    // set orientation of slider
    js.setOrientation(SwingConstants.VERTICAL);
    // set Font for the slider
    js.setFont(new Font("Serif", Font.ITALIC, 20));
    // add slider to panel
    jp.add(js);
    jp.add(jl);
    jf.add(jp);
    // set the text of label
    jl.setText("value of Slider is =" + js.getValue());
    // set the size of frame
    jf.setSize(300, 300);
    jf.show();
}

// if JSlider value is changed
public void stateChanged(ChangeEvent e)
{
    jl.setText("Slider is at =" + js.getValue());
}
}
```

Programming in JAVA

Output

PS C:\Users\ OneDrive\Documents\cap615\programs>javac slfont.java

Note: slfont.java uses or overrides a deprecated API.

Note: Recompile with -Xlint:deprecation for details.

PS C:\Users\ OneDrive\Documents\cap615\programs> java slfont

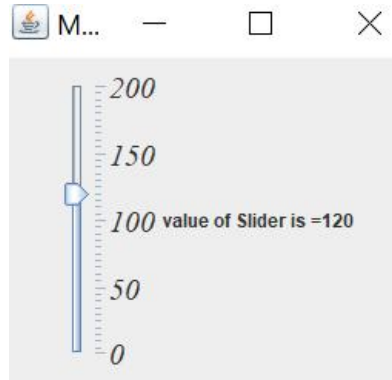


Figure 10: Slider with Major and Minor Ticks

Summary

- In swing the visual appearance is independent of internal representation, swing controls can be easily adjusted.
- JColorChooser is a class that provides a set of controls that allow a user to manipulate and select a color.
- The JSlider component is a swing component that allows the user to choose a value by moving a knob in a defined interval.
- SetMajorTickSpacing and setMinorTickSpacing in JSlider are used to adjust the number of values between the tick marks.
- JProgressBar is a visual representation of the progress of a task. It displays the percentage of a task that has been completed.
- A progress bar shows the progress of an event by displaying its percentage of completion and, in certain cases, a textual representation of that percentage.

Keywords

GUI: Graphical User Interface

JColorChooser: This component extends JComponent and allows the user to select a color.

getString(): It returns the string representation of the current value in the progress bar.

JSlider(): It produces a slider with a value of 50 as the initial point and a range of 0 to 100 as the range.

JSlider: A component that allows the user to select a value graphically by sliding a knob in a limited interval.

JComponent: This component is the root class for all Swing components, but not for the top-level containers.

javax.swing: A package in Java that provides a collection of lightweight components, that is, the components working similarly on all platforms.

Self Assessment

1. _____ sets the progress bar's minimum value to the value n.

- A. void setMinimum(int n)
 - B. int setMinimum(int n)
 - C. void SetMinimum(int n)
 - D. int SetMinimum(int n)
2. _____ creates a progress bar with specified minimum and maximum value.
- A. Jprogressbar(int min, int max)
 - B. JProgressBar(int min, int max)
 - C. JProgressbar(int min, int max)
 - D. jprogressbar(int min, int max)
3. _____ is a visible component to graphically display how much of a total task has completed.
- A. ProgressMonitor
 - B. JSlider
 - C. JProgressBar
 - D. None
4. Which of the following constructor is used to create the progress bar along with the mentioned orientation in its parameter.
- A. JProgressBar(int orientation)
 - B. JProgressbar(int orientation)
 - C. JProgressBar(int orientation, int min, int max)
 - D. JProgressBar(int orientation, int min, int max)
5. Which of the following get the progress bar's string representation of the current value.
- A. String GetString()
 - B. int getString()
 - C. Int getString()
 - D. String getString()
6. JSlider is a part of which of the following packages.
- A. Java Swing package.
 - B. Java *Lang* Package
 - C. Java *IO* Package
 - D. Java *Util* Package
7. Which of the following is not a JSlider constructor?
- A. JSlider(into)
 - B. JSlider(int min, int max)
 - C. JSlider(int min, int max, int value)
 - D. void setMinimum(int p)

Programming in JAVA

8. To set the extent of the slider, which of the following methods is used?
- A. SetExtent()
 - B. setextent()
 - C. setExtent()
 - D. All
9. Which of the following method is used to change the maximum value of a slider?
- A. setMaximum()
 - B. getMaximum()
 - C. Maximum()
 - D. useMaximum()
10. JSlider(int min, int max) is used to create a new slider with _____orientation and max and min value specified.
- A. Vertical
 - B. Horizontal
 - C. Mixed
 - D. None
11. The JColorChooser class is a pre-defined class of the _____package.
- A. Java.swing
 - B. javax.swing
 - C. java.lang
 - D. java.awt
12. JColorChooser inherits the _____ class.
- A. JComponent
 - B. JContainer
 - C. JSwing
 - D. All
13. Which of the followingconstructor creates a color chooser pane with white color as the default.
- A. jcolorchooser()
 - B. JColorChooser(color white)
 - C. JColorChooser()
 - D. ColorChooser()
14. Which of the followingmethod is used to return the current color from the color chooser dialog box.
- A. getColor()
 - B. setColor()
 - C. GetColor()
 - D. All

15. JColorChooser (Color col) is a color chooser pane created with the mentioned initial color col.
- A. False
 - B. True
 - C. Can't Say
 - D. May be

Answers for Self Assessment

- | | | | | |
|-------|-------|-------|-------|-------|
| 1. A | 2. B | 3. C | 4. A | 5. D |
| 6. A | 7. D | 8. C | 9. A | 10. B |
| 11. B | 12. A | 13. C | 14. A | 15. B |

Review Questions

1. "JcolorChooser helps in offering a control panel that is considered to allow a user to select a color".Comment on the statement with the help of an appropriate example.
2. Explain the various panes and methods used in JColorChooser class?
3. What is the significance of using JSlider? Discuss the various constructors of JSlider.
4. Write a program to show the specific string message on the progressbar.
5. " SetMajorTickSpacing and setMinorTickSpacing in JSlider are used to adjust the number of values between the tick marks" Justifying the statement with the help of an appropriate example.
6. Write a program to add a change listener to the slider and display the slider's value in a panel below the slider.



Further Readings

- Loy, M., Eckstein, R., Wood, D., Elliott, J., & Cole, B. (2002). *Java swing*. " O'Reilly Media, Inc."
- Cole, B., Eckstein, R., Elliott, J., Loy, M., & Wood, D. (2002). *Java Swing*. O'Reilly.
- Zukowski, J. (2006). *The definitive guide to Java Swing*. Apress.
- Topley, K. (2000). *Core Swing: advanced programming*. Prentice-Hall Professional.
- Robinson, M., & Vorobiev, P. (2004). *Swing*. Dreamtech Press.



Web Links

- <https://www.softwaretestinghelp.com/java/java-swing-tutorial/>
- <https://www.geeksforgeeks.org/java-swing-jcolorchooser-class/>
- <https://www.programcreek.com/java-api-examples/?api=javax.swing.JColorChooser>
- <https://examples.javacodegeeks.com/desktop-java/swing/java-swing-slider-example/>

Unit 10: Layouts

CONTENTS

Objectives

Introduction

10.1 Layout Manager

10.2 Border Layout

10.3 GridLayout

10.4 FlowLayout

10.5 BoxLayout

10.6 CardLayout

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

Objectives

After this unit you will be able to:

- Know the various types of Layouts.
- Learn the basic concept of Layout Manager and Border Layout.
- Understand and implement the various constructors and methods of Border Layout.
- Learn the basic concept of GridLayout, BoxLayout, and CardLayout.
- Explore and implement the various constructors and methods of GridLayout, BoxLayout, and CardLayout.
- Analyze and implement the various constructors and methods of FlowLayout.

Introduction

The arrangement of components within the container is referred to as layout. In other words, we can say that the components are placed in a specific location within the container. Layouts are used to format components on the screen, which is platform-independent. This means that the programs can be executed on multiple platforms. The Layout Manager is in charge of automatically laying out the controls.

10.1 Layout Manager

Layout Managers are used to putting the components in the correct order. It's an interface that implements the layout manager's classes. By determining the size and position of components within containers, the Layout managers allow us to control how visual components are ordered in GUI forms. Layout managers give programs a reliable and practical appearance, regardless of the platform, the screen size, or actions the user might take. The java development kit offers five classes and these classes implement the layout manager interface. Figure 1 illustrates these five classes.

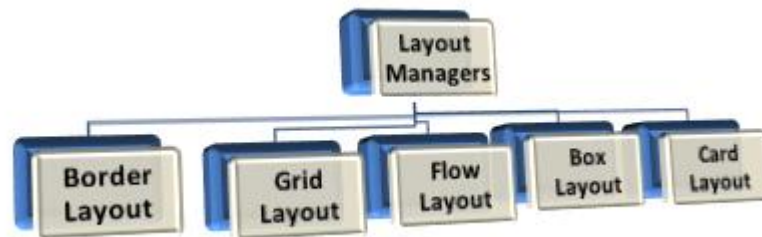


Figure 1: Types of Layouts

Setting the Layout Manager

JPanels and content panes are the only containers whose layout managers you need be concerned about. Unless you indicate otherwise when constructing the JPanel, each object is created to utilize a FlowLayout. BorderLayout is the default layout for content panes. If you don't like the default layout manager used by a panel or content pane, you can change it to something else. The FlowLayout and BorderLayout managers, on the other hand, are only useful for prototyping unless you're using JToolBar. The layout manager of a panel can be set using the JPanel constructor.



Example: `JPanel p = new JPanel (new BorderLayout ());`

The `setLayout` method can be used to set the layout manager for a container after it has been constructed.



Example: `Container contentPane = frame.getContentPane (); contentPane.setLayout (new FlowLayout ());`

Although layout managers are strongly recommended, you can do your layout without them. You may make a container utilize no layout manager by setting the layout attribute to null. You must describe the size and location of every component within that container using this method, known as absolute placement. Absolute positioning has the disadvantage of not adjusting adequately when the top-level container is resized. It also has trouble adapting to differences in users and systems, such as font sizes and locales.

Choosing a Layout Manager

Different strengths and disadvantages exist among layout managers. Here, we'll look at a few scenarios that can help us choose the best layout manager for the job.

CASE-1: A component must be displayed in as much space as possible.

Use GridLayout or BorderLayout if it's the lone component in its container. If you use BorderLayout, the space-hungry component must be placed in the center. GridBagLayout requires that you specify the component's constraints to `fill=GridBagConstraints.BOTH`. Another option is to use BoxLayout, which allows the component to set extremely large preferred and maximum sizes.

CASE-2: A few components must be displayed in a compact row at their original size.

Consider grouping the components in a JPanel and utilizing the default FlowLayout manager or the BoxLayout manager. This is when SpringLayout helps a lot.

CASE-3: A few components of the same size must be displayed in rows and columns.

GridLayout is ideal for this task.

CASE-4: You want to show a few components in a row or column, with various amounts of space between them, configurable alignment, and custom component sizes.

This is when BoxLayout works great.

CASE-5 You need to show aligned columns, such as in a form-like interface where a label column is used to describe text fields in a neighboring column.

For this, SpringLayout is an obvious choice. The makeCompactGrid function in the SpringUtilities class, which is used in various Tutorial examples, allows you to simply align several rows and columns of components.

CASE-6 You've got a complicated layout with a lot of moving parts.

To make layout easier, consider utilizing a flexible layout manager like GridBagLayout or SpringLayout, or grouping the components into one or more JPanels. If you go with the second option, each JPanel may utilize a distinct layout manager.

10.2 Border Layout

Border Layout is the default layout for Window, along with its children, Frame, and Dialog Border Layout. Border layout offers five different areas to hold components. These five areas are named after the four borders of the screen, namely North, South, East, and West. The remaining space is placed in the center area. Border Layout has one button in each area, before and after resizing. The center is the area in the middle.

Each region can only include one component and is designated by the letters NORTH, SOUTH, EAST, WEST, and CENTER.



Caution: When less than five components are placed in a container and BorderLayout is used, the empty component regions disappear. The remaining components then expand to fill the available space.

BorderLayout Constructors

BorderLayout Constructors are commonly used to create new border layouts with or without gaps between the components. The gap and gap arguments in the border layout constructors are used to set horizontal and vertical spacing between components. The following are the border layout constructors:

BorderLayout(): It will create a new BorderLayout with no gaps between the components.

BorderLayout(int, int): It creates a border layout with the provided gaps between components.

Methods of BorderLayout

The following are the various methods used in the border Layout:

toString(): It returns a string describing the current state of border layout.

getLayoutAlignmentX(Container parent): This method is used to return the X-axis layout alignment.

getLayoutAlignmentY(Container parent): This method returns the Y-axis layout alignment.

removeLayoutComponent(Component comp): Used to remove a component from the BorderLayout.

getVgap(): This method returns the vertical gap between the components.

getHgap(): The horizontal space between the components is returned by getHgap().

setHgap(int hgap): This method is used to adjust the horizontal gap between components.

setVgap(int vgap): This method is used to adjust the vertical gap between components.



Example:

```
//Implementation of border layout
import java.awt.*;
import javax.swing.*;

public class BL {
    JFrame jf;
    BL(){
        jf=new JFrame();
        //creation of button
        JButton jb1=new JButton("Welcome");
        JButton jb2=new JButton("Hello");
        JButton jb3=new JButton("Good");
        JButton jb4=new JButton("Day");
        JButton jb5=new JButton("Ahead");

        // Adding button to a frame
        jf.add(jb1,BorderLayout.NORTH);
        jf.add(jb2,BorderLayout.SOUTH);
        jf.add(jb3,BorderLayout.EAST);
        jf.add(jb4,BorderLayout.WEST);
        jf.add(jb5,BorderLayout.CENTER);

        // setting the size and visibility of frame.
        jf.setSize(300,300);
        jf.setVisible(true);
    }
    public static void main(String[] args) {
        new BL();
    }
}
```

Output

PS C:\Users\ OneDrive\Documents\cap615\programs> javac BL.java

PS C:\Users\ OneDrive\Documents\cap615\programs> java BL

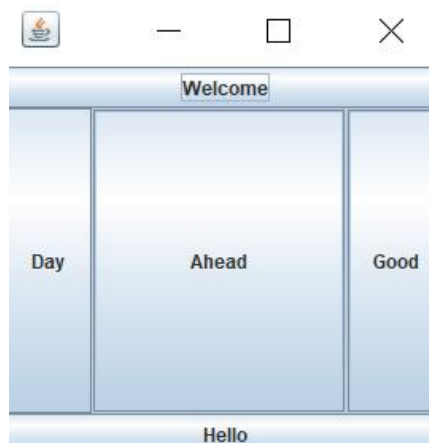


Figure 2: Output of Border Layout

10.3 GridLayout

In Graphic programming, layout managers are used to arranging components in a specified order. They're used to figure out how big a component is and where it should go in a container. Layout managers come in a variety of shapes and sizes. One of them is GridLayout. Grid Layout helps in the arrangement of components in rows and columns. This process of arrangement of components starts at the first row and column, then moves across the row until it is full, and then continues to the next row.

Grid Layout Manager allows the user to reposition or resize objects after adding or removing components. The GridLayout container is divided into rectangles of the same size. Each rectangle contains one of the components. Because each rectangle cell is the same size, it has a component that fills the full cell. When the user modifies or adjusts the container's size, the size of each rectangle adjusts as well.

GridLayout Constructors

GridLayout(): It creates a single-row grid layout with a default of one column per component.

GridLayout(int rw, int cl): It builds a grid layout with the number of rows and columns specified.

GridLayout(int rw, int cl, int hgap, int vgap): This function constructs a grid layout with the supplied number of rows and columns, as well as a horizontal and vertical gap.

GridLayout Methods

GridLayout is beneficial in Java when you wish to construct grids in a container with one condition: each of the grid's cells must be of equal size and uniformly distributed. The following are the various methods used in the GridLayout.

addLayoutComponent(String str, Component cmp): This method adds the supplied component to the layout with the specified name.

setColumns(int cl): Sets the number of columns in this layout to the specified value.

setHgap(int hgap): Sets the horizontal gap between components to the supplied value.

setRows(int rw): This method increases or decreases the number of rows in this layout.

setVgap(int vgap): Sets the vertical gap between components to the supplied value.

toString(): It returns a string representation of the values in this grid layout.



Example

// Implementation of GridLayout

```
import java.awt.*;
```

```
import javax.swing.*;
```

```
public class GridExample{
```

```
    JFrame jf;
```

```
    GridExample(){
```

```
        jf=new JFrame();
```

```
        JButton jb1=new JButton("A");
```

```
        JButton jb2=new JButton("B");
```

```
        JButton jb3=new JButton("C");
```

```
        JButton jb4=new JButton("D");
```

```
        JButton jb5=new JButton("E");
```

```

JButton jb6=new JButton("F");
JButton jb7=new JButton("G");
JButton jb8=new JButton("H");
JButton jb9=new JButton("I");
    //adding buttons to frame
jf.add(jb1);jf.add(jb2);
jf.add(jb3);
jf.add(jb4);jf.add(jb5);
jf.add(jb6);jf.add(jb7);
jf.add(jb8);
jf.add(jb9);
jf.setLayout(new GridLayout(3,3));
    //setting grid layout of 3 rows and 3 columns
jf.setSize(300,300);
jf.setVisible(true);
}
public static void main(String[] args) {
    newGridExample();
}
}

```

Output

PS C:\Users\ OneDrive\Documents\cap615\programs>javac GridExample.java

PS C:\Users\ OneDrive\Documents\cap615\programs> java GridExample

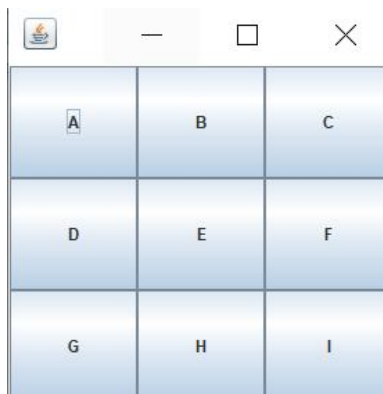


Figure 3: GridLayout Output

10.4 FlowLayout

Flow Layout is the default layout of the Panel class. When components are added to the screen, they flow from left to right based on the order added and the width of the applet. If many components have to be placed in a window, then they wrap to a new row.

In most cases, flow layouts are used to place buttons in a panel. It will arrange the buttons from left to right until there are no more buttons that will fit on the same line. Each line is in the middle.

Variables used in FlowLayout

LEFT: This value indicates that each component row should be justified to the left.

CENTER: This value indicates that the components in each row should be centered.

RIGHT: This value indicates that each component row should be justified to the right.

FlowLayout Constructors

The following table illustrates the various constructors of FlowLayout class along with their description:

Table 1: Constructors of FlowLayout

Constructor	Description
FlowLayout()	Creates a flow arrangement with centered alignment and a default horizontal and vertical gap of 5 units.
FlowLayout(int align)	It generates a flow layout with the specified alignment and a 5-unit horizontal and vertical gap as a default.
FlowLayout(int align, int hgap, int vgap)	Creates a flow layout with the supplied alignment and horizontal and vertical gaps

Methods used in FlowLayout

setTitle(String Text): This method is used to set the JFrame's title. The title to be set is specified as a string.

getAlignment(): Gets the alignment for this layout. Possible values are FlowLayout.LEFT, FlowLayout.RIGHT, or FlowLayout.CENTER.

setAlignment(int align): Sets the alignment for this layout. Possible values are FlowLayout.LEFT, FlowLayout.RIGHT, and FlowLayout.CENTER.

removeLayoutComponent(Component comp): Used to remove the component passed as an argument from the layout.



Example

//implementation of FlowLayout

```
import java.awt.*;
import javax.swing.*;

public class FlowLayoutExample
{
    JFrame jf;

    FlowLayoutExample()
    {
        jf = new JFrame();
        JButton jb1 = new JButton("1");
        JButton jb2 = new JButton("R");
        JButton jb3 = new JButton("A");
        JButton jb4 = new JButton("4");
        JButton jb5 = new JButton("B");
        jf.add(jb1);
        jf.add(jb2);
        jf.add(jb3);
    }
}
```

```

jf.add(jb4);
jf.add(jb5);
jf.setLayout(new FlowLayout(FlowLayout.RIGHT));
jf.setSize(300,300);
jf.setVisible(true);
}

public static void main(String[] args)
{
    newFlowLayoutExample();
}
}

```

Output

PS C:\Users\OneDrive\Documents\cap615\programs>javac FlowLayoutExample.java

PS C:\Users\OneDrive\Documents\cap615\programs> java FlowLayoutExample

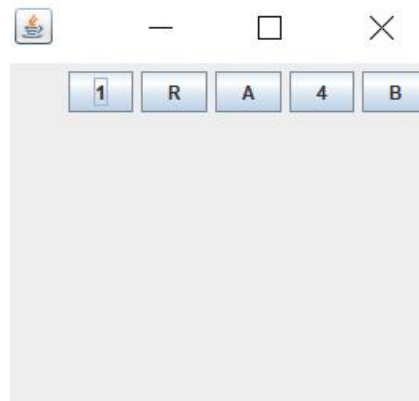


Figure 4: FlowLayout output



Task: Write a program to illustrate a flow layout manager in Java AWT. Drag the sides or corners of the displayed frame to demonstrate the working of the layout manager.

10.5 BoxLayout

BoxLayout is a common layout manager included with the Java platform. It aids in the horizontal or vertical arrangement of components within the container. Even if we enlarge the frame, the component arrangement will remain the same. i.e., even if the frame resizes, the vertical arrangement remains vertical. To put it another way, the contents of the container will not be wrapped. BoxLayout constructor, unlike other layout managers, gives the appropriate containers to the layout manager constructor. The next sections will go over the constructors, methods, and examples of the BoxLayout Class.

BoxLayout provides four constants to arrange the components either horizontally or vertically. They are as follows:

- **public static final int X_AXIS:** The components are aligned horizontally from left to right.
- **public static final int Y_AXIS:** The components are vertically aligned from top to bottom
- **public static final int LINE_AXIS:** Component alignment is comparable to how words are aligned in a line, and it is determined by the container's ComponentOrientation attribute. The components are positioned horizontally if the container's ComponentOrientation property is horizontal; otherwise, they are aligned vertically. There are two types of

horizontal orientations: left to right and right to left. Components are rendered from left to right if the container's `ComponentOrientation` property is set to left to right, and right to left if the container's `ComponentOrientation` property is set to right to left. The components are always shown from top to bottom in vertical orientations.

- **public static final int PAGE_AXIS:** The `ComponentOrientation` property of the container is used to align the components in the same way as text lines are aligned on a page. Components are positioned vertically if the container's `ComponentOrientation` attribute is horizontal; otherwise, components are aligned horizontally. There are two types of horizontal orientations: left to right and right to left. Components are rendered from left to right if the container's `ComponentOrientation` property is also set to left to right, and components are rendered from right to left if the container's `ComponentOrientation` property is set to right to left. The components are always shown from top to bottom in vertical orientations.

BoxLayout Constructor

The following constructor will be used to implement the `BoxLayout` class:

BoxLayout(Container c, int axis): It builds a box layout with the provided axis that arranges the components. Container and axis are the two arguments supplied, which aid in the creation of the container with the axis specified. The following are the two valid directions:

BoxLayout – from left to right.

X_AXIS

BoxLayout from top to bottom.

Y_AXIS

If we call `BoxLayout.Y-AXIS` we'll get the following output.

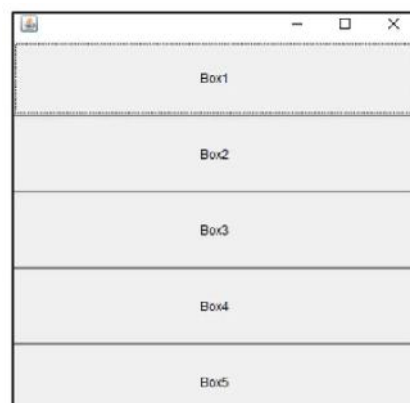


Figure 5: FlowLayout output

If we call `BoxLayout.X-AXIS` we'll get the following layout that looks like (one row).

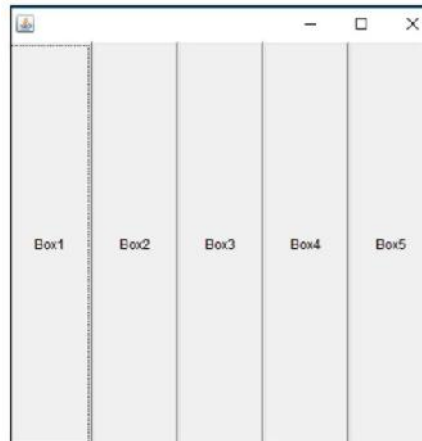


Figure 6:BoxLayout on X-AXIS

BoxLayout Methods

addLayoutComponent(Component cmp, Object obj): The goal of this method is to use the constraint object to add the indicated component to the container.

getLayoutAlignmentX(Container con): The container's alignment in the left to the right direction is returned by this method. X-Axis, to be precise. The value can range from 0 to 1, with 0 being the origin alignment, 1 representing the farthest alignment from the origin, and 0.5 representing the centered alignment.

getLayoutAlignmentY(Container con): The top-to-bottom alignment of the container is returned by this method. Y-Axis, for example. The value can range from 0 to 1, with 0 being the origin alignment, 1 representing the farthest alignment from the origin, and 0.5 representing the centered alignment.

maximumLayoutSize(Container con): This method provides the target container's maximum size for laying out the containers that are added to it.

minimumLayoutSize(Container con): This function returns the smallest size that the target container can use to layout any other containers.

removeLayoutComponent(Component cmp): This function removes the specified component from the layout of the container.

layoutContainer(Container tar): When the Abstract Window Toolkit calls this function, it lays up the container's layout (AWT).



Example

```
//BoxLayout example with different BoxLayout alignment settings
import javax.swing.BorderFactory;
import javax.swing.BoxLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import java.awt.Component;
import java.awt.FlowLayout;

public class BoxLayoutAlignExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("BoxLayout With Different Alignments");
```

```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
JButton jb1 = new JButton("OK");
JButton jb2 = new JButton("SUBMIT");
JButton jb3 = new JButton("DELETE");
JButton jb4 = new JButton("UPDATE");
JButton jb5 = new JButton("CANCEL");
JButton jb6 = new JButton("SAVE");
JPanel p1 = new JPanel();
JPanel p2 = new JPanel();
JPanel p3 = new JPanel();

p1.setBorder(BorderFactory.createTitledBorder("LEFT"));
p2.setBorder(BorderFactory.createTitledBorder("CENTER"));
p3.setBorder(BorderFactory.createTitledBorder("RIGHT"));

    // Set up the BoxLayout
BoxLayout layout1 = new BoxLayout(p1, BoxLayout.Y_AXIS);
BoxLayout layout2 = new BoxLayout(p2, BoxLayout.Y_AXIS);
BoxLayout layout3 = new BoxLayout(p3, BoxLayout.Y_AXIS);
p1.setLayout(layout1);
p2.setLayout(layout2);
p3.setLayout(layout3);

jb1.setAlignmentX(Component.LEFT_ALIGNMENT);
jb2.setAlignmentX(Component.LEFT_ALIGNMENT);
p1.add(jb1);
p1.add(jb2);

jb3.setAlignmentX(Component.CENTER_ALIGNMENT);
jb4.setAlignmentX(Component.CENTER_ALIGNMENT);
p2.add(jb3);
p2.add(jb4);

jb5.setAlignmentX(Component.RIGHT_ALIGNMENT);
jb6.setAlignmentX(Component.RIGHT_ALIGNMENT);
p3.add(jb5);
p3.add(jb6);

frame.setLayout(new FlowLayout());
frame.add(p1);
frame.add(p2);
frame.add(p3);
frame.pack();
frame.setVisible(true);
    }
}
```

Output

```
PS C:\Users\ OneDrive\Documents\cap615\programs>javac BoxLayoutAlignExample.java
```

```
PS C:\Users\ OneDrive\Documents\cap615\programs> java BoxLayoutAlignExample
```



Figure 7:BoxLayout Output

10.6 CardLayout

Card Layout, unlike other layouts that display the components of a container one at a time, functions as a deck of playing cards with only one card visible at a time, i.e. the topmost card. Every component in a container is treated as a Card, and the container itself is treated as a Stack of cards. Internally, the order of the cards in a container is determined. It is the initial component existing in the container that is seen when the container is displayed for the first time.

CardLayoutConstructors

CardLayout is a Java class that has a few constructors. Some of CardLayout's Constructors are listed below:

CardLayout():This constructor of the Java class CardLayout is used to build a new CardLayout with gaps between the individual components of size zero (0).

CardLayout(inthorizontalgap, intverticalgap): This Java constructor is used to generate a new CardLayout with the specified horizontal and vertical gaps between the components in the arguments. The horizontal gap between the components is denoted by Hgap, whereas the vertical gap is denoted by vgap.

Methods of CardLayout

next(Container parent):The procedure for advancing to the next card in the container.

previous(Container parent):The way for navigating to the previous card in the container.

first(Container parent):The mechanism for navigating to the first card in the container.

last(Container parent):The process for getting to the last card in the container.

show(Container parent, String name):The mechanism for navigating to the container with the supplied name.

getLayoutAlignmentX(Container parent):The alignment along the x-axis is returned by this method.

getLayoutAlignmentY(Container parent):The y-axis alignment is returned by this method.

getVgap():The method for determining the vertical space between the components.

addLayoutComponent(Component cm, Object cn):The method for adding a component to the internal table of components for a card layout.

getHgap():The method for determining the horizontal space between the components.

toString():The textual representation of this card arrangement is returned by this method.

removeLayoutComponent(Component cm):This method deletes the specified component from the card layout.



Example

```
// Program to show Examples of different methods of CardLayout.
```

```
import java.awt.*;
```

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;

public class CardLayoutMethod extends JFrame {
    private int currentCard = 1;
    private CardLayout cl;

    public CardLayoutMethod()
    {
        setTitle("Card Layout Example");
        setSize(300, 150);
        JPanel cPanel = new JPanel();
        cl = new CardLayout();
        cPanel.setLayout(cl);
        JPanel jp1 = new JPanel();
        JPanel jp2 = new JPanel();
        JPanel jp3 = new JPanel();
        JPanel jp4 = new JPanel();
        JLabel jl1 = new JLabel("First");
        JLabel jl2 = new JLabel("Second");
        JLabel jl3 = new JLabel("Third");
        JLabel jl4 = new JLabel("Fourth");
        jp1.add(jl1);
        jp2.add(jl2);
        jp3.add(jl3);
        jp4.add(jl4);
        cPanel.add(jp1, "1");
        cPanel.add(jp2, "2");
        cPanel.add(jp3, "3");
        cPanel.add(jp4, "4");
        JPanel buttonPanel = new JPanel();
        JButton fBtn = new JButton("First button");
        JButton nBtn = new JButton("Next button");
        JButton pBtn = new JButton("Previous button");
        JButton lBtn = new JButton("Last button");
        buttonPanel.add(fBtn);
        buttonPanel.add(nBtn);
        buttonPanel.add(pBtn);
        buttonPanel.add(lBtn);
        fBtn.addActionListener(new ActionListener()
        {
```

```
        public void actionPerformed(ActionEvent arg0)
        {
            // used first c1 CardLayout
            cl.first(cPanel);

            // value of currentcard is 1
            currentCard = 1;
        }
    });
    IBtn.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent arg0)
        {
            // used last c1 CardLayout
            cl.last(cPanel);
            // value of currentcard is 4
            currentCard = 4;
        }
    });
    nBtn.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent arg0)
        {
            if (currentCard < 4)
            {
                // increment the value of currentcard by 1
                currentCard += 1;
                // show the value of currentcard
                cl.show(cPanel, "" + (currentCard));
            }
        }
    });
    pBtn.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent arg0)
        {
            // if condition apply
            if (currentCard > 1) {
                // decrement the value
                // of currentcard by 1
            }
        }
    });
}
```

```

        currentCard -= 1;
        // show the value of currentcard
        cl.show(cPanel, "" + (currentCard));
    }
}

});
getContentPane().add(cPanel, BorderLayout.SOUTH);
getContentPane().add(buttonPanel, BorderLayout.NORTH);
}
public static void main(String[] args)
{
    CardLayoutMethod cl = new CardLayoutMethod();
    // Function to set default operation of JFrame.
    cl.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    cl.setVisible(true);
}
}

```

Output

PS C:\Users\ OneDrive\Documents\cap615\programs>javac CardLayoutMethod.java

PS C:\Users\ OneDrive\Documents\cap615\programs> java CardLayoutMethod



Figure 8:CardLayout Output



Lab Exercise: Write a program to illustrate a grid layout. The grid must contain five grids with different names.

Summary

- Layouts are used to format components on the screen.
- A layout manager is assigned to each container and is in charge of arranging the components within it.
- The layout manager arranges the components according to the layout manager's rules, property settings, and component limitations.
- A good appearance, regardless of the platform, screen size, or actions the user might take is given by the layout managers.
- The different types of layout managers are flow layout, border layout, grid layout, box layout, and card layout.
- The BorderLayout is used to vertically or horizontally organize the components.

- If there isn't enough horizontal space to hold all of the components in one row, Flow layout adds them to the next row, and so on.
- The layout manager positions all of the components within the container automatically, and if you don't use the layout manager, the default layout manager positions the components.
- When we want to arrange the components in five regions, we utilize BorderLayout.

Keywords

LayoutManager: The LayoutManager interface specifies which methods must be implemented by the class whose object will be used as a layout manager.

GridLayout: It arranges all of the elements in a grid of cells of equal size, adding them from left to right and top to bottom.

CardLayout: The components are organized in a deck with all of the same-sized cards and just the top card showing at any given moment.

previous(Container parent): The way for navigating to the previous card in the container.

getHgap(): The horizontal space between the components is returned by getHgap().

LEFT: This value indicates that each component row should be justified to the left.

BoxLayout: It arranges multiple components either **vertically or horizontally**, but not both.

Self Assessment

1. Positions the components into five regions: east, west, north, south, center
 - A. BorderLayout
 - B. CardLayout
 - C. GridLayout
 - D. FlowLayout
2. What are the names of the list layout managers in Java?
 - A. Flow Layout Manager
 - B. Grid Layout Manager
 - C. Box Layout Manager
 - D. All of the above
3. Which alignment directions may be used with BoxLayout?
 - A. Horizontal only.
 - B. Horizontal and Vertical.
 - C. Left aligned and Right aligned.
 - D. Centered and Justified.
4. What type of object determines where GUI components are placed in a container?
 - A. The layer organizer.
 - B. The component manager.
 - C. The frame manager.
 - D. The layout manager.

5. The default layout manager of a Frame is
- A. FlowLayout
 - B. GridLayout
 - C. BorderLayout
 - D. BoxLayout
6. How do you change the current layout manager for a container?
- A. Use the setLayout method.
 - B. Once created you cannot change the current layout manager of a component
 - C. Use the setLayoutManager method
 - D. None
7. The default orientation to display the progress bar is?
- A. CENTER
 - B. TOP
 - C. HORIZONTAL
 - D. VERTICAL
8. Which of the following is the default Layout Manager for an Applet?
- A. FlowLayout
 - B. BorderLayout
 - C. GridLayout
 - D. CardLayout
9. The _____ is used to arrange the components in a line, one after another.
- A. BorderLayout
 - B. GridLayout
 - C. FlowLayout
 - D. CardLayout
10. Which of the following are the constructors of FlowLayout?
- A. FlowLayout()
 - B. FlowLayout(int align)
 - C. FlowLayout(int align, intHorizontalGap, intVerticalGap)
 - D. All of the above.
11. Which of the following is not a method of FlowLayout?
- A. setTitle(String Text)
 - B. getAlignment()
 - C. SetAlignment(int align)
 - D. setAlignment(int align)

12. Arranges the components horizontally
- A. BorderLayout
 - B. CardLayout
 - C. GridLayout
 - D. FlowLayout
13. Which layout manager works by positioning components in rows and columns?
- A. BorderLayout
 - B. FlowLayout
 - C. GridLayout
 - D. BoxLayout
14. When the user changes or adjusts the size of the container in GridLayout, the size of each _____ changes accordingly.
- A. Rectangle
 - B. Square
 - C. Circle
 - D. None
15. Arranges the components into a grid
- A. BorderLayout
 - B. CardLayout
 - C. GridLayout
 - D. FlowLayout

Answers for Self Assessment

- | | | | | |
|-------|-------|-------|-------|-------|
| 1. A | 2. D | 3. B | 4. D | 5. C |
| 6. A | 7. C | 8. A | 9. C | 10. D |
| 11. C | 12. D | 13. C | 14. A | 15. C |

Review Questions

1. "The Java development kit offers five classes and these classes implement the layout manager interface". Discuss.
2. "Card Layout, unlike other layouts that display the components of a container one at a time" justify the answer with an appropriate example.
3. Discuss the various scenarios that can help us choose the best layout manager.
4. Elucidate BorderLayout along with its various methods.
5. "BoxLayout provides four constants to arrange the components either horizontally or vertically" what are they? Explain each component with the help of an example?

6. Enlighten the significance of using GridLayout. Discuss the various constructors and methods used by GridLayout.



Further Readings

JADHAV, T. (2020). Advance Java.

Arnold, K., Gosling, J., & Holmes, D. (2005). The Java programming language. Addison Wesley Professional.

Liang, Y. D. (2003). Introduction to Java programming. Pearson Education India.

Sharan, K. (2014). Beginning Java 8 APIs, Extensions, and Libraries: Swing, JavaFX, JavaScript, JDBC, and Network Programming APIs. Apress.

Sharan, K., Sharan, K., & Anglin. (2018). Java APIs, Extensions, and Libraries. Apress.



Web Links

<https://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html>

https://www.tutorialspoint.com/awt/awt_layouts.htm

<https://www.c-sharpcorner.com/UploadFile/9a9e6f/layout-managers-in-java-part-1/>

<https://ecomputernotes.com/java/awt-and-applets/gridlayoutjavaexample>

http://www.java2s.com/Tutorial/Java/0240__Swing/WhatistheBorderLayout.htm

Unit 11: Managing Data using JDBC

CONTENTS

Objectives

Introduction

11.1 JDBC

11.2 JDBC Driver

11.3 JDBC Architecture

11.4 Database Connectivity

11.5 CRUD Operations

11.6 Connection Interface

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

Objectives

- Learn the basic concept of JDBC and understand the different types of JDBC drivers.
- Know and analyze the different architectural components of JDBC.
- Understand the various steps required for database connectivity.
- Implement database connectivity and CRUD operations.

Introduction

The JDBC API is a Java API that allows you to access any type of tabular data, including data from a Relational Database. JDBC enables you to create Java applications that handle the following three programming tasks:

- Connect to a data source, like a database
- Send queries and update statements to the database
- Retrieve and process the results received from the database in answer to your query

JDBC was created as a client-side API that allowed a Java client to interface with a database. With JDBC 2.0, which added an optional package that supported server-side JDBC connections, that changed. Since then, every major JDBC release has included improvements to both the client-side (java.sql) and server-side (java.sql) packages (javax.sql).

11.1 JDBC

JDBC stands for Java Database Connectivity and is an Application Programming Interface (API) for connecting Java applications to databases. It can communicate with a variety of databases, including Oracle, MS Access, My SQL, and SQL Server. It's also known as a platform-agnostic interface between a relational database and Java programming. It enables a Java program to execute SQL statements and obtain database results.

Programming in JAVA

The JDBC API is made up of classes and methods that are used to connect to, read from, write to, and store data in a database.

Use of JDBC

- Enterprise applications built with the JAVA EE platform must communicate with databases to store application-specific data.
- Before JDBC, the ODBC API was the database API for connecting to the database and running queries.
- However, the ODBC API makes use of the ODBC driver, which is written in C. (i.e. platform-dependent and unsecured).
- We can use JDBC API to handle database using Java program and can perform the following activities:
 - Connect to the database.
 - Execute queries and update statements to the database.
 - Retrieve the result received from the database.

Purpose of JDBC

Enterprise applications built with the JAVA EE platform must communicate with databases to store application-specific data. As a result, communicating with a database necessitates effective database connectivity, which can be accomplished with the ODBC (Open database connectivity) driver. This driver is used in conjunction with JDBC to connect or communicate with a variety of databases, including Oracle, MS Access, Mysql, and SQL Server.

11.2 JDBC Driver

JDBC Driver is a type of software that allows a Java program to interact with a database. For connecting with your database server, JDBC drivers implement the stated interfaces in the JDBC API.



Example: JDBC drivers, allow you to open database connections and interact with them by delivering SQL or database commands and then getting the results in Java.

The `java.sql` package that comes with JDK has several classes that have their behavior defined but are implemented by third-party drivers. The `java.sql.Driver` interface is implemented by third-party database drivers.

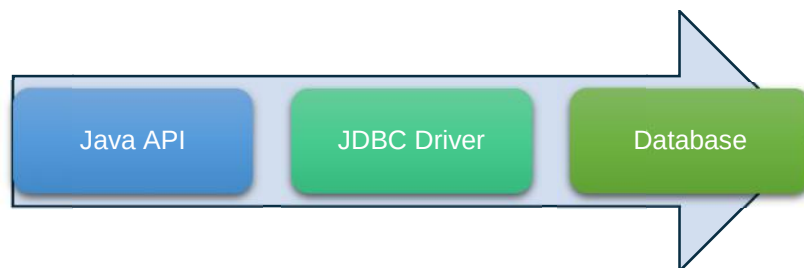


Figure 1: JDBC Driver

Types of JDBC Drivers

There are 4 types of JDBC drivers:

1. JDBC-ODBC bridge driver
2. Native-API driver (partially java driver)
3. Network Protocol driver (fully java driver)

4. Thin driver (fully java driver)

JDBC-ODBC bridge driver

The ODBC driver is used by the JDBC-ODBC bridge driver to connect to the database. JDBC method calls are converted to ODBC function calls by the JDBC-ODBC bridge driver. Because of the thin driver, this is now discouraged.

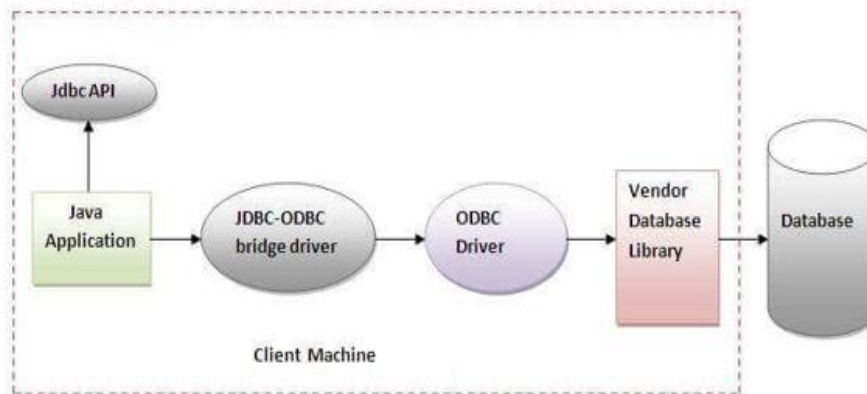


Figure 2: JDBC-ODBC bridge driver

Advantages

- It is simple to use.
- It's simple to connect to any database.

Disadvantages

- Performance suffers as a result of the conversion of JDBC method calls to ODBC function calls.
- On the client's PC, the ODBC driver must be installed.

Native-API driver

The Native API driver makes use of the database's client-side libraries. The driver translates JDBC method calls to database API native calls. It is not written in Java.

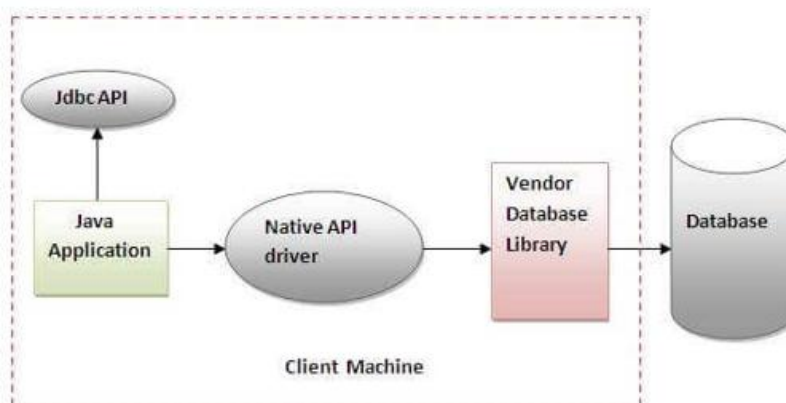


Figure 3: Native-API driver

Advantage

- Performance is improved over the JDBC-ODBC bridge driver.

Disadvantages

Programming in JAVA

- The Native driver must be installed on each client machine, which is a disadvantage.
- On the client machine, the Vendor client library must be installed.

Network Protocol driver

The Network Protocol driver makes use of middleware (application server) to convert JDBC calls into the vendor-specific database protocol, either directly or indirectly. It's written entirely in Java.

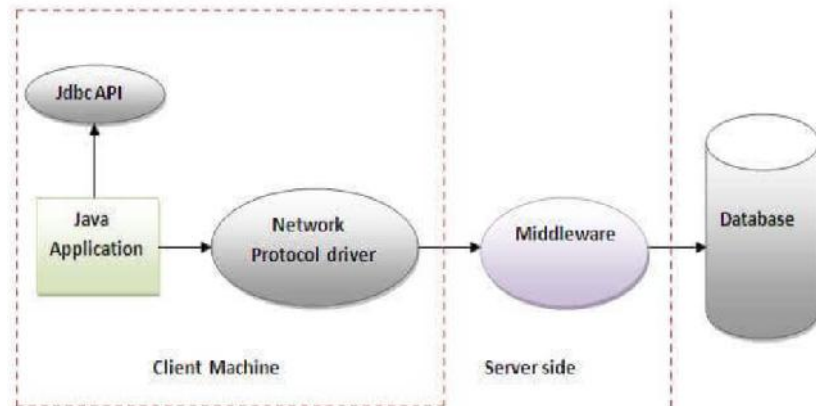


Figure 4: Network Protocol driver

Advantage

- Because the application server can do numerous activities such as auditing, load balancing, and logging, no client-side libraries are required.

Disadvantages

- The client machine must have network support.
- In the middle tier, database-specific coding is required.
- Network Protocol driver maintenance is costly since it necessitates database-specific coding in the middle tier.

Thin driver

JDBC calls are converted directly into the vendor-specific database protocol via the thin driver. That is why it is referred to as a "thin driver." It is entirely written in the Java programming language.

Advantages

Outperforms all other drivers in terms of performance.

There is no software required on either the client or server-side

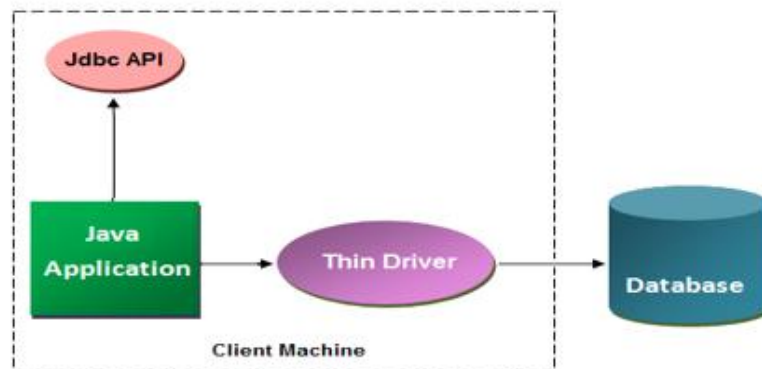


Figure 5: Thin driver

Disadvantage

Drivers are reliant on the database, which is a disadvantage.

Which Driver should be Used

- If you are accessing one type of database, such as Oracle, Sybase, or IBM, the preferred driver type is 4.
- If your Java application is accessing multiple types of databases at the same time, type 3 is the preferred driver.
- Type 2 drivers are useful in situations, where a type 3 or type 4 driver is not available yet for your database.
- The type 1 driver is not considered a deployment-level driver and is typically used for development and testing purposes only.

Components of JDBC

The following components are required to interface with the database using JDBC.

DriverManager: The `java.sql` package's `DriverManager` class manages many types of JDBC drivers. This class is responsible for loading the driver classes. Furthermore, once a new connection is established, it selects and loads the appropriate driver from the previously loaded drivers.



Note: Beginning with JDBC 4.0, the drivers in the CLASSPATH will be automatically loaded.

Driver: The Driver interface is in charge of setting up database connections. A `Connection` object representing a database connection is returned by the `connect` method. The database server's communications are handled via this interface. Driver items are rarely interacted with directly. Instead, you utilize `DriverManager` objects, which are used to handle this type of object. It also hides the essential details of working with Driver objects.

Connection: This is the interface that contains all of the methods for contacting a database. The connection object represents the communication context, i.e., the connection object is the sole way to communicate with the database.

The core of the JDBC API is the `Connection` interface. Most of the methods in the `Connection` interface can be divided into three categories:

- Getting database information
- Creating database statements
- Managing database transactions

Getting database information

The `Connection` interface's `getMetaData` method returns a `DatabaseMetaData` object that describes the database. You can receive a list of all database tables and look at each one's definition. In most database applications, metadata isn't very valuable, but it's ideal for creating database explorer tools that gather information about the database.

Creating database statements

Statements are used to run database commands. The `Statement`, `PreparedStatement`, and `CallableStatement` are the three sorts of statements. Each of these instructions operates in a slightly different manner, but the final result is always the same: a database command is executed.

Managing database transactions

Every statement you issue is usually treated as a single database transaction. However, there are situations when you want to combine numerous statements into a single transaction. The `auto-commit` flag in the `Connection` class indicates whether or not transactions should be committed automatically.

Statement: To execute SQL statements to the database, you use objects built using this interface. In addition to performing stored procedures, certainly derived interfaces accept parameters.



Note: When you close a `Connection`, it automatically closes any open statements or result sets.

ResultSet: You can use the ResultSet interface to get data from a SQL query. The ResultSet interface is most commonly used to read data, however, as of JDBC version 2.0, you can also update and delete records. If you merely want to read the results, use the next method to move to the next row, then use one of the several get methods to receive the data.

These objects store data acquired from a database after utilizing Statement objects to run a SQL query. It functions as an iterator, allowing you to cycle through the data it contains.



Example:

```
ResultSet results = stmt.executeQuery(  
    "select last_name, age from Person);  
while (results.next())  
{  
    String lastName = results.getString("last_name");  
    int age = results.getInt("age");  
}
```

SQLException: This class handles any errors that occur in a database application.

11.3 JDBC Architecture

JDBC API should be used if you wish to create a Java application that connects with a database. A driver is an implementation of the API; different vendors provide different drivers; you must use the appropriate driver for the database you need to interface with. The driver manager is in charge of loading and managing the driver.

Application: It is a java applet or a servlet that communicates with a data source.

The JDBC API: The JDBC API allows Java programs to execute SQL statements and retrieve results. Some of the important classes and interfaces defined in JDBC API are as follows:

- DriverManager
- Driver
- Connection
- Statement
- PreparedStatement
- CallableStatement
- ResultSet
- SQL data

DriverManager: It plays an important role in the JDBC architecture. It uses some database-specific drivers to effectively connect enterprise applications to databases.

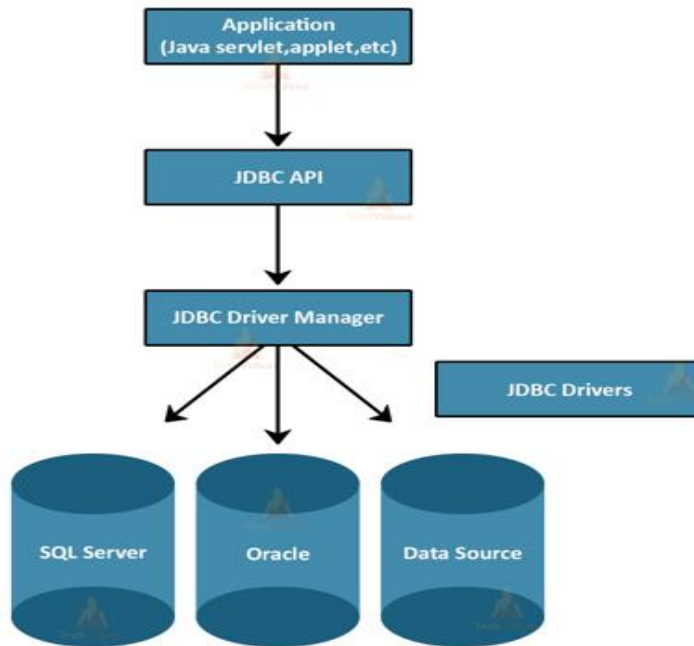


Figure 6: JDBC Architecture

JDBC drivers: To communicate with a data source through JDBC, you need a JDBC driver that intelligently communicates with the respective data source.

Types of JDBC Architecture

To access a database, the JDBC design uses a two-tier and three-tier approach.

Two-tier model: In a two-tier paradigm, the application communicates directly with the data source. The JDBC driver is responsible for connecting the data source to the application. When a user sends a query to the data source, the user receives a direct response to those inquiries.

The data source can be on a different system, and that machine is connected to the user machine via a client-server paradigm, in which the machine that sends the query is the client, and the one that sends the results of those queries is the server.

Three-tier model: In this architecture, the user's queries are delivered to middle-tier services, which then send the commands back to the data source. The responses to those requests are routed back to the intermediate tier, where they are provided to the user again.

11.4 Database Connectivity

There are 5 steps to connect any java application with the database using JDBC. The following are the steps to follow for database connectivity:

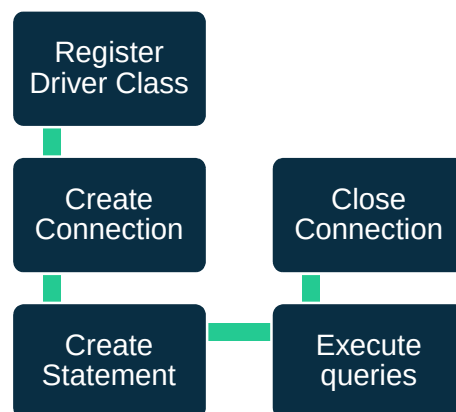


Figure 7: Database Connectivity steps

Register the driver class: The driver class is registered using the Class class's `forName()` method. This technique is used to load the driver class dynamically.



Example: `Class.forName("MySQL.jdbc.driver.MySQLDriver");`

Create the connection object: The `DriverManager` class's `getConnection()` method is used to establish a database connection.



Example: `Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","password");`

Create the Statement object: To generate a statement, use the `Connection` interface's `createStatement()` method. The statement's object is in charge of doing queries on the database.



Example: `Statement stmt=con.createStatement();`

Execute the query: The `executeQuery()` function of the `Statement` interface is used to run database queries. This function returns a `ResultSet` object that may be used to get all of a table's records.



Example: `ResultSet rs=stmt.executeQuery("select * from Tablename"); while(rs.next()){ System.out.println(rs.getInt(1)+" "+rs.getString(2)); }`

Close the connection object: By closing the connection object statement and `ResultSet` will be closed automatically. The `close()` method of the `Connection` interface is used to close the connection.



Example: `con.close();`

11.5 CRUD Operations

The CRUD operation can be defined as user interface conventions that allow view, search, and modify information through computer-based forms and reports. CREATE, READ, UPDATE, and DELETE are the essential functions of persistent storage, and CRUD is an acronym for them. To retrieve and return data from a database, CRUD operations can employ forms or an interface view. The following table illustrates the various CRUD operations along with their description:

Table 1: CRUD Operations

Operation	Description
CREATE	This is a form view to create a new record in the database
READ	Reads the table records based on the primary key within the input parameter.
UPDATE	Updates the content of the table based on the specified primary key for a record.

Creating a Database

Here we are going to discuss how we can create a database using NetBeans. The following steps you need to follow for the creation of a database.

To create a new database, right-click on Java DB again. From the menu that appears, select Create Database:

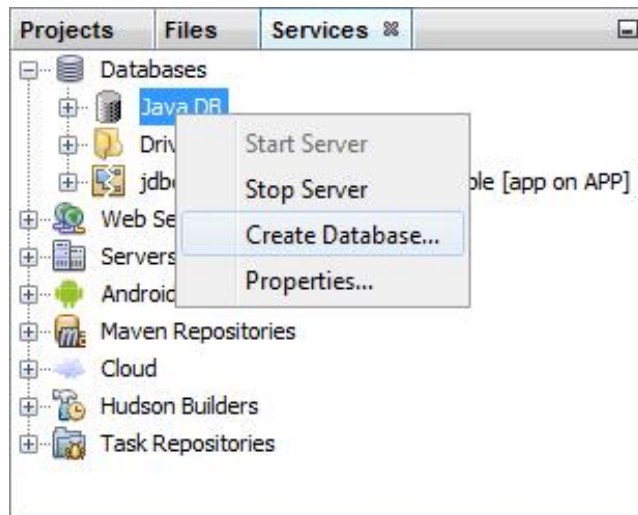


Figure 8: database Creation Option

When you click on Create Database, you'll see a dialogue box appear:

Type a name for your database in the first box. Call it DBStudent. Type any User Name and Password.

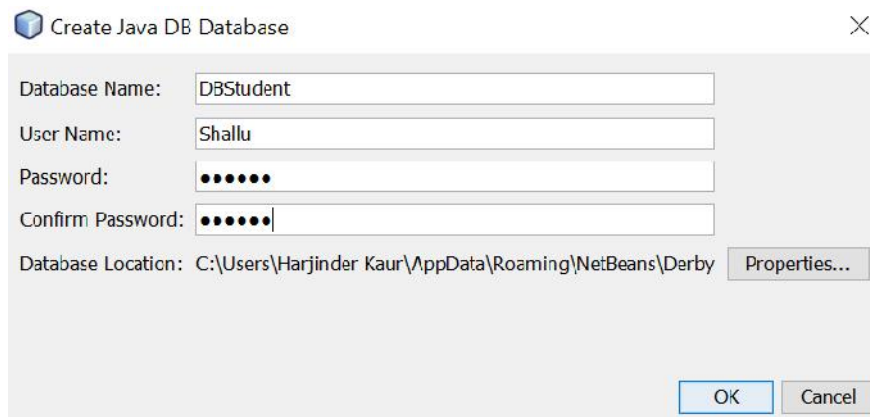


Figure 9: Required Information for Database Creation

Click OK to create your database. It should then appear on the list:

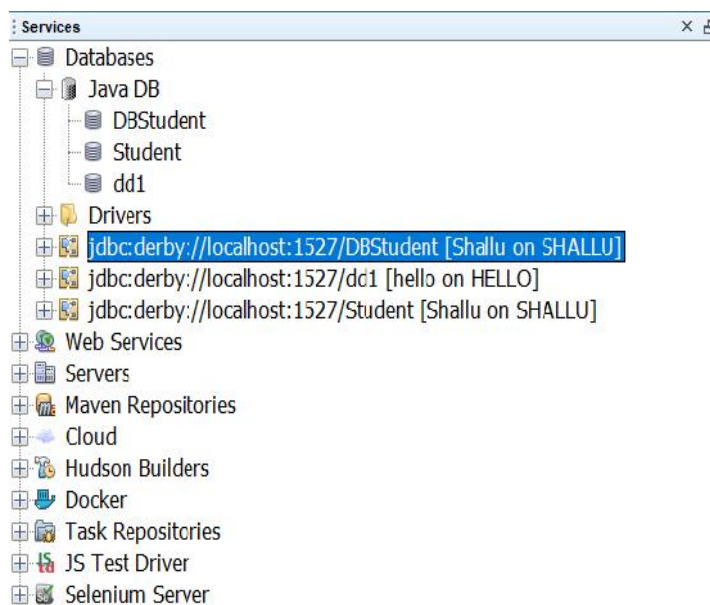


Figure 10: Created Database path

Creating a Table in the Database

Now that the database has been created, you need to create a table in the database. To do so, right-click on your database. From the menu that appears select **Connect**:

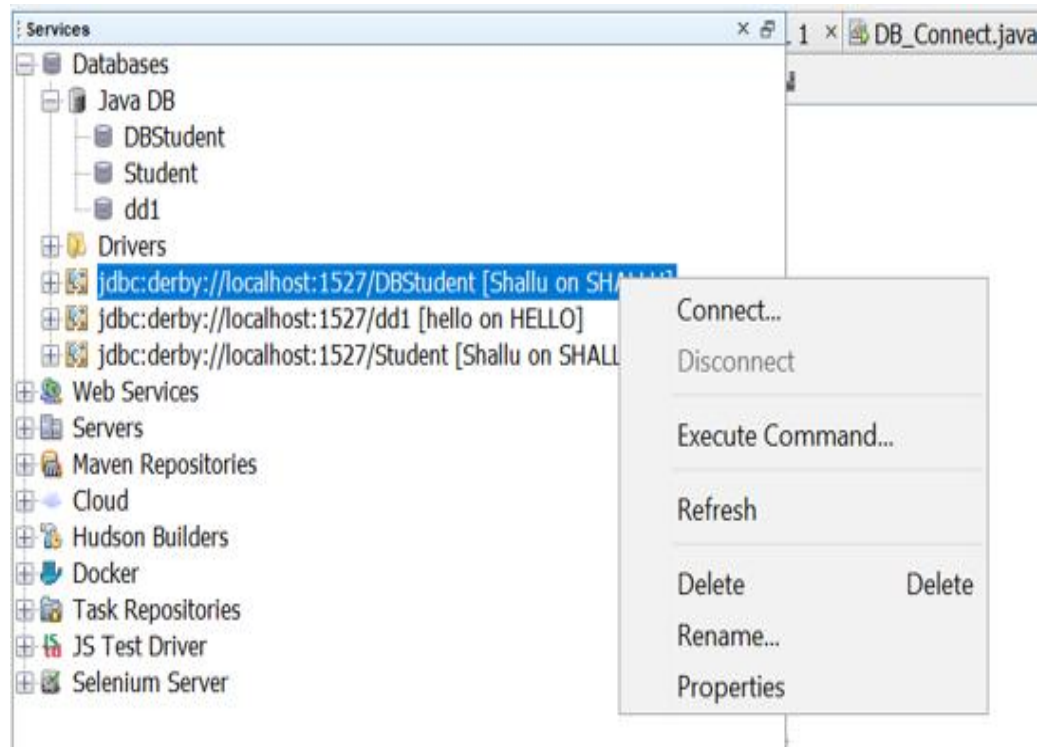


Figure 11: Connection to Database

When a connection is made, you'll see some default folders for Tables, Views, and Procedures. To create a new table in your database, right-click the **Tables** folder. From the menu that appears, select **Create Table**:

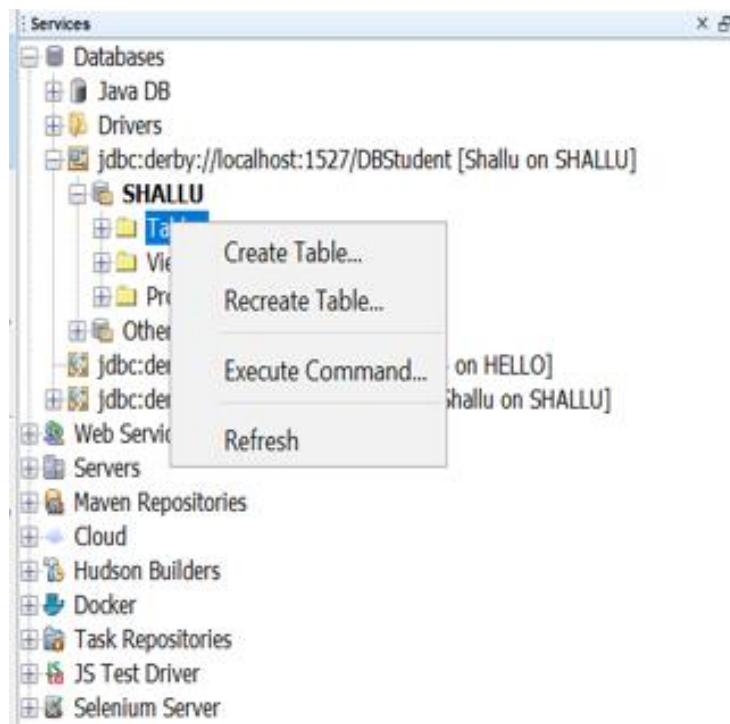


Figure 12: Creation of Table

When you click on **Create Table**, a dialogue box appears:

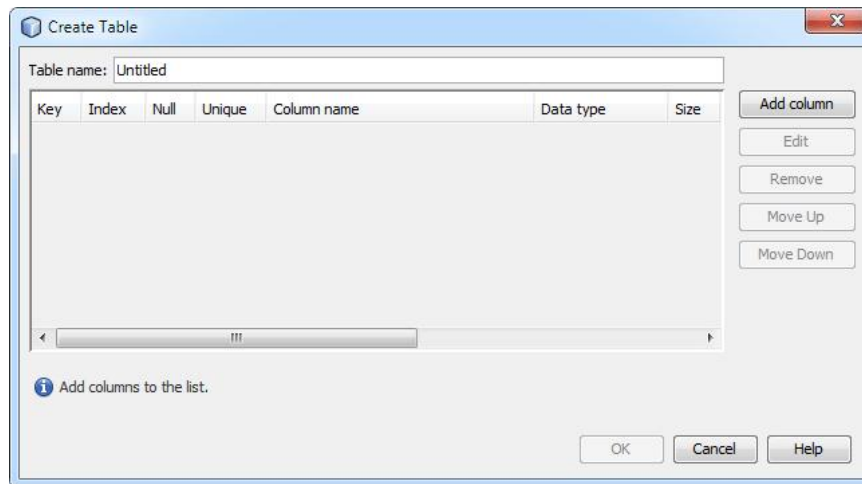


Figure 13: Interface for Adding Attributes

Here you can give the table name you want to create and add attributes into it using add column command.

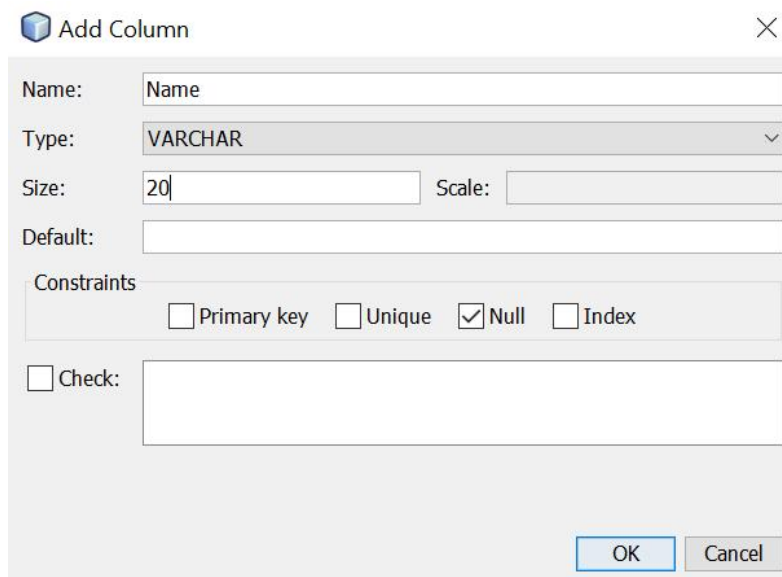


Figure 14: Application of different types and Constraints

Where The NAME is the name of the column in the table, like ID, First_Name, etc. The TYPE is the DATA TYPE, Integer, VARCHAR, etc. Click the dropdown list to see more. Then check or uncheck the CONSTRAINTS boxes as indicated.

Insert Record(s)

Press CTRL+Tab to exit data entry mode from the table. Press CTRL+0 to set NULL value and CTRL+1 to set DEFAULT value for a given column.

#	NAME	ADDRESS	CONTACT
1	<NULL>	<NULL>	<NULL>

Show SQL Add Row Remove OK Cancel

Figure 15: Insertion of Records

Figure 16: Designed Interface for CRUD implementation

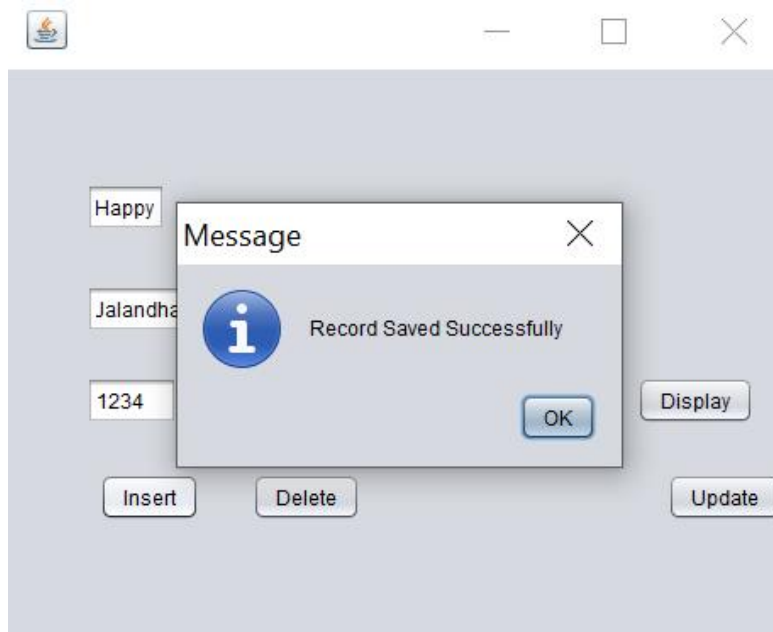


Figure 17: Implementation of insert Operation

SELECT * FROM SHAIJU.TEST...			
Max. rows: 100 Fetched Rows: 3 Matching Rows:			
#	NAME	ADDRESS	CONTACT
1	Rahul	USA	444
2	hello	123	root
3	Happy	1234	Jalandhar

Figure 18: Result of Insert Operation



Example

// Implementation of CRUD Operations

Database Connectivity

```

package Empl;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.ResultSet;
import java.util.logging.Logger;
import javax.swing.JOptionPane;

public class Works extends javax.swing.JFrame {
    Connection con;
    Statement stmt;
    ResultSets;

    public Works() {
        initComponents();
        DoConnect();
    }

```

```

    }
    public void DoConnect( )
    {
        try
        {
            String host="jdbc:derby://localhost:1527/DBStudent";
            String uname="Shallu";
            String pwd="Shallu";
            con=DriverManager.getConnection(host, uname, pwd);
        }
    catch(SQLException e)
    {
        JOptionPane.showMessageDialog(this, e.getMessage());
    }
}
@SuppressWarnings("unchecked")
// <editor-fold defaultstate="collapsed" desc="Generated Code">
private void initComponents() {
    jTextField1 = new javax.swing.JTextField();
    jTextField2 = new javax.swing.JTextField();
    jTextField3 = new javax.swing.JTextField();
    jButton1 = new javax.swing.JButton();
    jButton2 = new javax.swing.JButton();
    jButton3 = new javax.swing.JButton();
    jButton4 = new javax.swing.JButton();
    setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);

    jTextField1.setText("Name");
    jTextField1.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent evt) {
            jTextField1ActionPerformed(evt);
        }
    });
    jTextField2.setText("Address");

    jTextField3.setText("Contact");

    jButton1.setText("Insert");
    jButton1.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent evt) {
            jButton1ActionPerformed(evt);

```

```

    }
});

jButton2.setText("Delete");
jButton2.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        jButton2ActionPerformed(evt);
    }
});

jButton3.setText("Update");
jButton3.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        jButton3ActionPerformed(evt);
    }
});

jButton4.setText("Display");
jButton4.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        jButton4ActionPerformed(evt);
    }
});

private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    try
    {
stmt=con.createStatement();
        String name=jTextField1.getText();
        String mob=jTextField2.getText();
        String add=jTextField3.getText();
        String sql="Insert into TBSTU values('"+(name)+"','"+(add)+"','"+(mob)+"')";
stmt.executeUpdate(sql);
JOptionPane.showMessageDialog(this, "Record Saved Successfully");
        jTextField1.setText("");
        jTextField2.setText("");
        jTextField2.setText("");
    }
catch(SQLException e)
    {
JOptionPane.showMessageDialog(this, e.getMessage());

```

```

    }

}

private void jButton4ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    try
    {
        stmt=con.createStatement();
        String sql="Select * from TBstu";
        rs=stmt.executeQuery(sql);
        while(rs.next())
        {
            jTextField1.setText(rs.getString("NAME"));
            jTextField2.setText(rs.getString("ADDRESS"));
            jTextField3.setText(rs.getString("Contact"));
        }
    }
    catch(SQLException e)
    {
        JOptionPane.showMessageDialog(this, e.getMessage());
    }

}

private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    try
    {
        stmt=con.createStatement();
        String mob=jTextField2.getText();
        String sql="Delete from TBstu where Contact = '" + (mob) + "'";
        stmt.executeUpdate(sql);
        JOptionPane.showMessageDialog(this, "Record Deleted Successfully");
        jTextField1.setText("");
        jTextField2.setText("");
        jTextField3.setText("");
    }
    catch(SQLException e)
    {
        JOptionPane.showMessageDialog(this, e.getMessage());
    }
}

```

```

    }

}

private void jButton3ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    try
    {
stmt=con.createStatement();
        String name=jTextField1.getText();
        String mob=jTextField2.getText();
        String add=jTextField3.getText();

        String sql="Update TBstu set NAME='"+ (name)+" ', ADDRESS='"+ (add)+" ' where Mob = '"
+ (mob) + "'";
stmt.executeUpdate(sql);
OptionPane.showMessageDialog(this, "Record updated Successfully");
        jTextField1.setText("");
        jTextField2.setText("");
        jTextField3.setText("");
    }
catch(SQLException e)
    {
OptionPane.showMessageDialog(this, e.getMessage());
    }
}

private void jTextField1ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
}

/**
 * @param args The command-line arguments
 */
public static void main(String args[]) {
java.awt.EventQueue.invokeLater(new Runnable() {
    public void run() {
        new Works().setVisible(true);
    }
});
}

// Variables declaration - do not modify
private javax.swing.JButton jButton1;
private javax.swing.JButton jButton2;
private javax.swing.JButton jButton3;

```

```

private javax.swing.JButton jButton4;
private javax.swing.JTextField jTextField1;
private javax.swing.JTextField jTextField2;
private javax.swing.JTextField jTextField3;
// End of variables declaration
}

```

11.6 Connection Interface

A Connection is a session between a java application and a database. The Connection interface is a factory of Statement, PreparedStatement, and DatabaseMetaData i.e. object of Connection can be used to get the object of Statement and DatabaseMetaData. The Connection interface provide many methods for transaction management like commit(), rollback() etc.

Methods of the Connection interface

The following table shows the various methods used by the connection interface along with their description:

Connection Interface Methods

Method	Function
public Statement createStatement()	It creates a statement object that can be used to execute SQL queries.
public Statement createStatement(int resultSetType,intresultSetConcurrency)	Creates a Statement object that will generate ResultSet objects with the given type and concurrency.
public void setAutoCommit(boolean status)	It is used to set the commit status.By default it is true.
public void commit()	It is used to save the changes made since the previous commit/rollback is permanent.
public void rollback()	Drops all changes made since the previous commit/rollback.
public void close()	It closes the connection and Releases JDBC resources immediately.

Summary

- JDBC is a software tool that makes a connection between a standard database and a Java application that intends to use that database, also known as an application programming interface.
- The first layer of the JDBC architecture is the JDBC driver, which has direct communication with the data storage devices.
- JDBC drivers are client-side adapters that translate Java program requests into a protocol that the database management system can understand.
- The CRUD operation can be defined as user interface conventions that allow view, search, and modify information through computer-based forms and reports.
- A result set is a table of rows and columns that contains the results of executing an SQL query.

- The connection interface represents the connection with a specific database.

Keywords

JDBC: JDBC stands for Java Database Connectivity, which is a standard Java API for database-independent connectivity between the Java programming language, and a wide range of databases.

JDBC Drivers: A JDBC driver is a collection of Java classes that enables you to connect to a certain database.

Statement: A Statement is what you use to execute queries and updates against the database.

ResultSet: When you perform a query against the database you get back a ResultSet.

Application: It is a java applet or a servlet that communicates with a data source.

Driver: This interface handles the communications with the database server.

SQL: Structured Query Language is a standardized language that allows you to perform operations on a database, such as creating entries, reading content, updating content, and deleting entries

Self Assessment

1. What must be the first characters of a database URL?
 - A. db,
 - B. db:
 - C. jdbc,
 - D. jdbc:
2. To transfer data between a database and an application written in the Java programming language, the JDBC API provides which of these methods?
 - A. Methods on the ResultSet class for retrieving SQL SELECT results as Java types.
 - B. Methods on the PreparedStatement class for sending Java types as SQL statement parameters.
 - C. Methods on the CallableStatement class for retrieving SQL OUT parameters as Java types.
 - D. All mentioned above
3. The method executes a simple query and returns a single Result Set object
 - A. executeUpdate()
 - B. executeQuery()
 - C. execute()
 - D. noexecute()
4. How many JDBC driver types does Sun define?
 - A. One
 - B. Two
 - C. Three
 - D. Four
5. A Connection is the session between java application and _____.

- A. Table
 - B. Database
 - C. Row
 - D. Column
6. Which of the following is not a method of Connection interface.
- A. public void commit()
 - B. public void rollback()
 - C. public void call()
 - D. public void close()
7. A _____ object is used when an application plans to specify parameters to your SQL queries.
- A. statement
 - B. prepared statement
 - C. data source
 - D. parameterized statement
8. A _____ object in JDBC controls the connection to the database.
- A. Connection
 - B. DataSource
 - C. Statement
 - D. CallableStatement
9. Which of the following method is used to perform DML statements in JDBC?
- A. executeResult()
 - B. executeQuery()
 - C. executeUpdate()
 - D. execute()
10. How many transaction isolation levels provide the JDBC through the Connection interface?
- A. 3
 - B. 4
 - C. 7
 - D. 2
11. The method on the result set that tests whether or not there remains at least one unfetched tuple in the result set, is said to be
- A. Fetch method
 - B. Current method
 - C. Next method
 - D. Access method
12. The _____ package contains classes that help in connecting to a database, sending SQL statements to the database, and processing the query request.

- A. connection.sql
- B. db.sql
- C. pkg.sql
- D. java.sql

13. What are the major components of the JDBC?

- A. DriverManager, Driver, Connection, Statement, and ResultSet
- B. DriverManager, Driver, Connection, and Statement
- C. DriverManager, Statement, and ResultSet
- D. DriverManager, Connection, Statement, and ResultSet

14. Which of the following is the advantage of using the JDBC connection pool?

- A. Slow performance
- B. Using more memory
- C. Using less memory
- D. Better performance

15. What is, in terms of JDBC, a DataSource?

- A. A DataSource is the basic service for managing a set of JDBC drivers
- B. A DataSource is the Java representation of a physical data source
- C. A DataSource is a registry point for JNDI-services
- D. A DataSource is a factory of connections to a physical data source

Answers for Self Assessment

- | | | | | |
|-------|-------|-------|-------|-------|
| 1. D | 2. D | 3. B | 4. D | 5. B |
| 6. C | 7. B | 8. A | 9. C | 10. B |
| 11. C | 12. D | 13. A | 14. D | 15. D |

Review Questions

- 1) What are the important components of JDBC?
- 2) Explain the various JDBC drivers.
- 3) With the help of an example elucidate the various steps of database connectivity.
- 4) Describe the JDBC architecture in detail.
- 5) Explain the significance of using a connection interface along with its various methods.
- 6) Write down the step-by-step process of creation of the database and table.
- 7) Write a note on the following:
 - a) Two-tier Architecture.
 - b) Three-tier Architecture.



Further Readings

Patel, P., & Moss, K. (1997). *Java database programming with JDBC* (pp. 5-8). Coriolis Group Books.

Khannedy, E. K. (2011). MySQL dan Java Database Connectivity. *Bandung: StripBandung*.

Bales, D. (2002). *Java Programming with Oracle JDBC*. " O'Reilly Media, Inc."

Reese, G. (2000). *Database Programming with JDBC and JAVA*. " O'Reilly Media, Inc."

Parsian, M. (2006). *JDBC Metadata, MySQL, and Oracle recipes: a problem-solution approach*. Apress.



Web Links

<https://www.baeldung.com/java-jdbc>

<https://www.simplilearn.com/tutorials/java-tutorial/java-jdbc>

<https://www.ibm.com/docs/en/informix-servers/12.10?topic=started-what-is-jdbc>

<https://docs.oracle.com/javase/tutorial/jdbc/basics/index.html>

<https://www.codejava.net/java-se/jdbc/jdbc-tutorial-sql-insert-select-update-and-delete-examples>

Unit 12: More on JDBC

COTENTS

Objectives

Introduction

12.1 Statement Interface

12.2 Result Set interface

12.3 DatabaseMetaData Interface

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

Objectives

After this unit you will be able to:

- Understand and implement Statement and ResultSet Interface
- Learn and implement Resultset Interface
- Know the concept, methods of DatabaseMetaData Interface and its implementation

Introduction

Java database connectivity (JDBC) specifies interfaces and classes that enable database applications to be written in Java by establishing database connections. JDBC facilitates SQL statement execution and provides basic SQL capability. Because thin clients run Java, applets placed in Web sites provide downloadable JDBC scripts that enable remote database access. When a Java application needs a database connection, one of the `DriverManager.getConnection()` methods are used to create a JDBC connection. The following are the various use cases used in JDBC:

Query the database: Reading data from a database is one of the most prevalent usage cases. Querying a database is the process of reading data from it.

Query the database metadata: Querying the database metadata is another typical use case. The database metadata is made up of details about the database.



Example: Data on the tables defined, the columns in each table, the data formats, and so on.

Update the database: Updating the database is another popular JDBC use case. Writing data to the database is what updating it entails. Adding new records or altering (updating) existing ones, in other words.

Perform transactions: Another prominent use case is transactions. A transaction is a single action that combines several updates and perhaps queries. Either all of the activities are carried out, or none of them are carried out.

**Did you Know?**

Java Database Connectivity (JDBC) is a Java SE edition application program interface (API) that standardizes and simplifies the process of connecting Java programs to external relational database management systems (RDBMS).

12.1 Statement Interface

The Statement interface provides methods for executing database queries. The statement interface is a ResultSet factory, which means it provides factory methods for getting a ResultSet object. To update or query the database, the Statement object runs a standard SQL statement. The java.sql.Statement interface is part of the JDBC API and defines a standard abstraction for statement objects that are implemented by the JDBC driver.

The statement interface in Java is used to generate SQL basic statements and provides methods for executing database queries. The following are the different sorts of statements used in JDBC:

- Create Statement
- Prepared Statement
- Callable Statement

Create a Statement: You can create the object for this interface from the connection interface. It's typically utilized for general-purpose database access and comes in handy when running static SQL commands.

Syntax:

```
Statement statement = connection.createStatement();
```

The required SQL statement is sent as a query, and the Statement object delivers a set of results. Because we can't construct interface objects, we can utilise the connection object's createStatement() function to get a Statement object. A single connection object can be used to produce numerous objects.



Example:

```
package db_console;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

/**
 *
 * @author Harjinder Kaur
 */

public class DB_Connect {

    /**
     * @param args The command-line arguments
     * @throws java.sql.SQLException
     */
}
```

```

public static void main(String[] args) throws SQLException {
    // TODO code application logic here
    try {
        String host= "jdbc:derby://localhost:1527/Student";
        String uname="Shallu";
        String pass="Shallu";
        Connection con=DriverManager.getConnection(host, uname, pass);
        Statement stmt=con.createStatement();
        String s="select * from stu";
        ResultSets=stmt.executeQuery(s);
        rs.next();
        int r=rs.getInt("Roll");
        String Sname=rs.getString("Name");
        String Scity=rs.getString("City");
        System.out.println("Roll=" + r + "\n" + "name=" + Sname + "\n" + "city=" + Scity);
    }
    catch(SQLException e)
    {
        System.out.println(e.getMessage());
    }
}
}

```

Output

run:

Roll=1

name=Geeta

city=Delhi

BUILD SUCCESSFUL (total time: 0 seconds)

Prepared Statement: The term "prepared statement" refers to a recompiled SQL statement that can be used several times. This accepts SQL queries with parameters. In this case, "?" is used instead of the parameter; however, the parameter can be passed dynamically by utilizing the PREPARED STATEMENT methods at run time.

Callable Statement: Callable Statements are stored procedures, which are a collection of statements that we compile in the database for a specific task. They are useful when dealing with multiple tables in a complex scenario and instead of sending multiple queries to the database, we can send the required data to the stored procedure and reduce the logic that is executed on the database server. The JDBC API's Callable Statement interface aids in the execution of stored procedures.



Example: `CallableStatement stmt=con. prepareCall("{call myprocedure(?,?)}");`

Methods of Statement Interface

public boolean execute(String url): This approach applies to any SQL statement (eg. Select, Insert, Update, etc.). The result of this procedure is a boolean value. If you're not sure which method to use (executeQuery() or executeUpdate()), apply the execute() method.

public ResultSet executeQuery(String url): This method is used to retrieve data from the database using the Select a statement. A ResultSet object is returned by this function.

public int executeUpdate(String url): If you wish to make changes to your database, use the executeUpdate() method. This method returns an int value indicating the number of rows that have been affected.

public int[] executeBatch(): This method is used to execute the batch of commands. It returns an integer array.

12.2 Result Set interface

A cursor referring to a table row is maintained by the ResultSet object. The cursor is initially positioned before the first row. However, we can make this object go forward and backward by supplying TYPE_SCROLL_INSENSITIVE or TYPE_SCROLL_SENSITIVE to the createStatement(int,int) method, and we can also make it updatable by:

Syntax

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATABLE);
```

The data is returned in a result set by SQL statements that read data from a database query. The conventional technique to select rows from a database and view them in a result set is to use the SELECT command.

A database query's result set is represented via the ResultSet interface. A cursor is maintained in a ResultSet object that points to the current row in the result set. The row and column data included in a ResultSet object are referred to as a "result set."

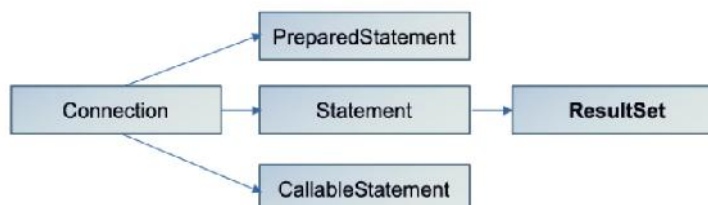


Caution: If one ResultSet's reading is interleaved with the reading of another, each must have been generated by different Statements.

The following are the several types of ResultSet interface methods:

- **Navigational methods:** It is used to move the pointer around.
- **Get methods:** Used to see the data in the columns of the current row where the cursor is pointing.
- **Update methods:** Update methods are used to update the data in the current row's columns. After that, the updates can be applied to the underlying database.

Based on the attributes of the ResultSet, the cursor can be moved. When the corresponding Statement that generates the ResultSet is generated, these characteristics are specified.



Java ResultSet Class Hierarchy

Types of ResultSet

The various RSTypes are listed here. If you don't specify a ResultSet type, you'll get one that's TYPE_FORWARD_ONLY by default.

Unit 12: More on JDBC

ResultSet.TYPE_FORWARD_ONLY: In the result set, the cursor can only move ahead.

ResultSet.TYPE_SCROLL_INSENSITIVE: The cursor can move forward and backward in the result set, and the result set is unaffected by changes to the database made by others after it was formed.

ResultSet.TYPE_SCROLL_SENSITIVE: The cursor can go forward and backward, and the result set is sensitive to changes to the database made by others after it was formed.



Note: There is no size() or length() method for Resultsets in Java.

Methods used in ResultSet

The following table shows the various methods used in ResultSet:

Method	Description
public boolean next()	used to move the cursor to the one row next from the current position.
public boolean previous()	used to move the cursor to the one row previous from the current position.
public boolean first()	used to move the cursor to the first row in the result set object.
public boolean last()	used to move the cursor to the last row in the result set object.
public boolean absolute(int row)	used to move the cursor to the specified row number in the ResultSet object.
public boolean relative(int row)	used to move the cursor to the relative row number in the ResultSet object, it may be positive or negative.
public int getInt(int columnIndex)	used to return the data of specified column index of the current row as int.
public int getInt(String columnName)	used to return the data of specified column name of the current row as int.
public String getString(int columnIndex)	used to return the data of specified column index of the current row as String.
public String getString(String columnName)	used to return the data of specified column name of the current row as String.



Example:

```
package db_console;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
```

Programming in JAVA

```

/**
 *
 * @author Harjinder Kaur
 */
public class DB_Connect {

    public static void main(String[] args) throws SQLException {
        // TODO code application logic here

        String host= "jdbc:derby://localhost:1527/Student";
        String uname="Shallu";
        String pass="Shallu";
        Connection con=DriverManager.getConnection(host, uname, pass);
        Statement stmt=con.createStatement();
        String s="select * from stu";
        ResultSets=stmt.executeQuery(s);
        while(rs.next())
        {
            int r=rs.getInt("Roll");
            String Sname=rs.getString("Name");
            String Scity=rs.getString("City");

            System.out.println("Roll=" + r + "\n" + "name=" + Sname + "\n" + "city=" + Scity);
        }
    }
}

```

Output

```

Roll=1
name=Geeta
city=Delhi
Roll=2
name=Rahul
city=Jalandhar
Roll=3
name=Hello
city=LDH
Roll=6
name=Atul
city=Ludhiana
Roll=15
name=Harry

```

city=Bombay

12.3 DatabaseMetaData Interface

Metadata is a term that refers to data about data. The DatabaseMetaData interface provides methods for obtaining information about the database you've connected to, such as the database name, database driver version, and maximum column length, among other things.

For instance, you can examine what tables are defined in the database, as well as what fields each table has and whether or not certain features are available. The Connection interface's getMetaData() method returns a DatabaseMetaData object.

Syntax:

public DatabaseMetaData getMetaData() throws SQLException.

Methods of DatabaseMetaData Interface

The following are the various methods used along with DatabaseMetaData interface:

- **getDriverName():** It returns the name of the JDBC driver.
- **getDriverVersion():** It returns the version number of the JDBC driver.
- **getUserName():** It returns the username of the database.
- **getDatabaseProductName() :** It returns the product name of the database.
- **getNumericFunctions():** Retrieves the list of the numeric functions available with this database.
- **getStringFunctions():** Retrieves the list of the numeric functions available with this database.
- **getSystemFunctions():** Retrieves the list of the system functions available with this database.
- **getDatabaseProductVersion() :** It returns the product version of the database.
- **getTables(String catalog, String schemaPattern, String tableNamePattern, String[]):** It returns the description of the tables of the specified catalog. The table type can be TABLE, VIEW, ALIAS, SYSTEM TABLE, SYNONYM, etc.
- **getTimeDateFunctions():** Retrieves the list of the time and date functions available with this database.
- **getURL():** Retrieves the URL for the current database.



Example

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package db_console;
import java.sql.Connection;
import java.sql.DatabaseMetaData;
import java.sql.DriverManager;
import java.sql.ResultSet;
```

```

import java.sql.SQLException;

/**
 *
 * @author Harjinder Kaur
 */
public class DBMetaExa {
    public static void main(String args[]){
    try{
        //Class.forName("oracle.jdbc.driver.OracleDriver");

        Connectioncon=DriverManager.getConnection("jdbc:derby://localhost:1527/DBStudent","Shallu","Shallu");
        DatabaseMetaDatadbmd=con.getMetaData();
        System.out.println("Driver Name: "+dbmd.getDriverName());
        System.out.println("Driver Version: "+dbmd.getDriverVersion());
        System.out.println("UserName: "+dbmd.getUserName());
        System.out.println("Database Product Name: "+dbmd.getDatabaseProductName());
        System.out.println("Database Product Version: "+dbmd.getDatabaseProductVersion());
        System.out.println("Numeric functions:" +dbmd.getNumericFunctions());
        String tab[]={"TABLE"};
        ResultSets=dbmd.getTables(null,null,null,tab);
        while(rs.next()){
            System.out.println(rs.getString(3));
        }
        con.close();
    }catch(SQLException e){ System.out.println(e);}
    }
}

```

Output

Driver Name: Apache Derby Network Client JDBC Driver

Driver Version: 10.11.1.2 - (1629631)

UserName: Shallu

Database Product Name: Apache Derby

Database Product Version: 10.10.2.0 - (1582446)

Numeric

functions:ABS,ACOS,ASIN,ATAN,ATAN2,CEILING,COS,COT,DEGREES,EXP,FLOOR,LOG,LOG10,MOD,PI,RADIANS,RAND,SIGN,SIN,SQRT,TAN

TBSTU

Summary

- JDBC stands for Java Database Connectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.

- The JDBC API uses a driver manager and database-specific drivers to provide transparent connectivity to heterogeneous databases.
- PreparedStatements have the added benefit of preventing SQL injection attacks.
- The DatabaseMetaData interface provides methods for obtaining information about the database you've connected to.
- The row and column data included in a ResultSet object are referred to as a "result set."
- The statement interface is a ResultSet factory, which means it provides factory methods for getting a ResultSet object.
- If you don't specify a ResultSet type, you'll get one that's TYPE_FORWARD_ONLY by default.

Keywords

Prepared Statement: The term "prepared statement" refers to a recompiled SQL statement that can be used several times.

Statement: The Statement interface provides methods for executing database queries.

getDriverVersion(): It returns the version number of the JDBC driver.

getInt(String columnName): It is used to return the data of the specified column name of the current row as int.

ResultSet.TYPE_SCROLL_SENSITIVE: The cursor can go forward and backward, and the result set is sensitive to changes to the database made by others after it was formed.

Update methods: Update methods are used to update the data in the current row's columns.

Callable Statement: Callable Statements are stored procedures, which are a collection of statements that we compile in the database for a specific task.

Self Assessment

1. PreparedStatement Object in JDBC is used to execute_____queries.
 - A. executable
 - B. simple
 - C. high level
 - D. parameterized
 2. Which of the following is the statement that permits the storage of large objects in the database?
 - A. Callable statement
 - B. Object statement
 - C. Prepared statement
 - D. None
 3. The PreparedStatement interface is a subinterface of_____.
- A. Statement
 - B. Connection
 - C. Layout
 - D. Network

4. Which method is used for an SQL statement that is executed frequently?
- A. prepare call
 - B. prepare statement
 - C. create statement
 - D. none of the above
5. Which of the following is not the method of PreparedStatement interface?
- A. setInt(int, int)
 - B. setString(int, string)
 - C. setFloat(int, float)
 - D. setLength(int)
6. ResultSetMetaData interface is useful because it provides methods to get metadata from the ResultSet object.
- A. True
 - B. False
 - C. Cant'say
 - D. May be
7. Which of the following is a method of ResultSetMetaData Interface?
- A. getColumnCount()
 - B. getcolumncount()
 - C. getColumnCount()
 - D. getColumncount()
8. We can use the _____ method of the connection object to obtain Statement object.
- A. createStatement
 - B. insertStatement
 - C. updateStatement
 - D. All of the above
9. Which of the following is/are the methods of statement interface?
- A. execute(String url)
 - B. executeQuery(String url)
 - C. executeUpdate(String url)
 - D. All of the above
10. The object of ResultSet maintains a cursor pointing to a _____ of a table.
- A. Column
 - B. Row
 - C. Table
 - D. None of the above

11. The interface `ResultSet` has a method, `getMetaData()`, that returns a/an

- A. Tuple
- B. Value
- C. Object
- D. Result

12. Which of the following means that the `ResultSet` can only be read?

- A. `ResultSet.CONCUR_READ_ONLY`
- B. `ResultSet.CONCUR_UPDATABLE`
- C. `ResultSet.READ_ONLY`
- D. `ResultSet.UPDATABLE`

13. Which of the following under JDBC statements?

- A. Statement
- B. Prepared Statement
- C. Callable Statement
- D. All of the above

14. `DatabaseMetaData` is used to

- A. execute DML statement
- B. execute Select statement only
- C. get inside information of database
- D. None of given

15. Which of the following returns the username of the database.

- A. `getuserName()`
- B. `getUserName()`
- C. `getusername()`
- D. All of the above

Answers for Self Assessment

- | | | | | |
|-------|-------|-------|-------|-------|
| 1. D | 2. C | 3. A | 4. B | 5. D |
| 6. A | 7. C | 8. A | 9. D | 10. B |
| 11. C | 12. A | 13. D | 14. C | 15. B |

Review Questions

1. Define the concept of JDBC. Discuss in detail the various types of statements available in JDBC.

2. How Statement is different from the PreparedStatement interface? What is the benefit of using PreparedStatement in Java?
3. What is the significance of using DatabaseMetaData interface? Explain the various methods of DatabaseMetaData interface with the help of an appropriate example.
4. Explain the concept of ResultSet. Discuss the different types of ResultSet by giving the example of each.
5. What happens if we call resultSet.getInt(0) when Select query result just have one column of integer type with the help of an appropriate example?
6. With example elucidate the different methods used to view the data using ResultSet. Explain the various methods of ResultSet?
7. Write a note on the following:
 - a) PreparedStatement.
 - b) ResultSet.
 - c) JDBC



Further Readings

Patel, P., & Moss, K. (1997). Java database programming with JDBC (pp. 5-8). Coriolis Group Books.

Keogh, J. (2003). J2ME: The complete reference. Osborne.

Reese, G. (2003). Java Database Best Practices: Persistence Models and Techniques for Java Database Programming. " O'Reilly Media, Inc."

Reese, G. (2000). Database Programming with JDBC and JAVA. " O'Reilly Media, Inc."

Thomas, T. M., & Books, M. T. (2002). Java data access: JDBC, JNDI, and JAXP. M & T Books.



Web Links

<https://www.mysqltutorial.org/jdbc-overview/>

<https://docs.oracle.com/javase/tutorial/jdbc/overview/index.html>

<http://tutorials.jenkov.com/jdbc/resultset.html>

<https://www.softwaretestinghelp.com/jdbc-resultset-tutorial/>

<https://www.geeksforgeeks.org/establishing-jdbc-connection-in-java/>

<https://www.ibm.com/docs/en/b2b-integrator/5.2?topic=l-java-database-connectivity-jdbc-adapter-v523-later>

Unit 13: Network Programming

CONTENTS

Objectives

Introduction

13.1 Networking

13.2 Socket Class

13.3 ServerSocket Class

13.4 URL class

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

Objectives

- Know the basic concept of Java Networking and its various terminologies
- Understand the various classes used in networking
- Learn the concept of Socket, ServerSocketclass, its methods, and applications

Introduction

Java networking refers to the concept of linking two or more computing devices to exchange resources. At the application layer, Java software communicates via the network. All Java networking classes and interfaces can be found in the java.net package.

Two protocols are supported by the java.net package. The following are the details:

TCP: TCP (Transmission Control Protocol) is a protocol that allows two programs to communicate reliably. TCP is most commonly used with TCP/IP, which stands for Transmission Control Protocol/Internet Protocol.

UDP: The User Datagram Mechanism (UDP) is a connectionless protocol for sending data packets between applications.



Note: Java networking is mostly used for resource sharing and centralized software management.

13.1 Networking

Network programming refers to writing programs that execute across multiple devices (computers), in which the devices are connected via a network. Java encapsulates classes and interfaces to allow low-level communication details. Java Networking is a concept of connecting two or more computing devices so that we can share resources. Java program communicates over the network at the application layer. The java.net package is useful for all the Java networking classes and interfaces.

Java Networking Terminologies

The widely used Java networking terminologies are:

- **IP Address:** IP address is a unique number assigned to a node of a network e.g. 192.168.0.1. It is composed of octets that range from 0 to 255. It is a logical address that can be changed.
- **Protocol:** A protocol is a set of rules that is followed for communication.



Example: TCP,FTP,Telnet,SMTP

- **Port Number:** The port number is used to uniquely identify different applications. It acts as a communication endpoint between applications. The port number is associated with the IP address for communication between two applications.
- **MAC Address:** MAC (Media Access Control) address is a unique identifier of NIC (Network Interface Controller). A network node can have multiple NICs but each with a unique MAC address.



Example, an ethernet card may have a **MAC** address of 00:0d:83::b1:c0:8e.

- **Connection-oriented and Connection-less protocol:** In the connection-oriented protocol, acknowledgment is sent by the receiver. So it is reliable but slow. The



Example of a connection-oriented protocol is TCP. But, in the connection-less protocol, acknowledgment is not sent by the receiver. So it is not reliable but fast.



Example of a connection-less protocol is UDP.

- **Socket:** A **socket** in Java is one endpoint of a two-way communication link between two programs running on the network. A **socket** is bound to a port number so that the TCP layer can identify the application that data is destined to be sent to. Now that you know various terminologies used in Java Networking, let's move further and understand some of the important classes that it supports.

InetAddress

Inet Address is used to encapsulate both the numerical IP address and the domain name for that address. It can handle both IPv4 and Ipv6 addresses. The Figure 1 depicts the subclasses of the Inet Address class. Three commonly used Inet Address factory methods are.

- ✓ static InetAddress getLocalHost() throws UnknownHostException.
- ✓ static InetAddress getByName (String hostname) throws UnknownHostException.
- ✓ static InetAddress[] getAllByName (String hostname) throws UnknownHostException.

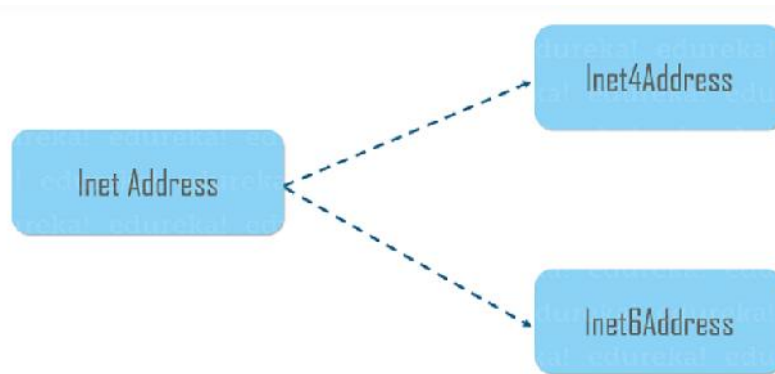


Figure 1: Types of InetAddress



Example

```
import java.net.*;

class Demo
{
    public static void main(String[] args) throws UnknownHostException
    {
        InetAddress address = InetAddress.getLocalHost();
        System.out.println(address);
        address = InetAddress.getByName("www.google.com");
        System.out.println(address);
    }
}
```

Output

```
PS C:\Users\Harjinder Kaur\OneDrive\Documents\cap615>javac Demo.java
```

```
PS C:\Users\Harjinder Kaur\OneDrive\Documents\cap615> java Demo
```

```
LAPTOP-QBKBNCCF/192.168.1.4
```

```
www.google.com/142.250.206.132
```

13.2 Socket Class

The Socket is the foundation of modern networking, a socket allows a single computer to serve many different clients at once. Socket establishes a connection through the use of a port, which is a numbered socket on a particular machine. Socket communication takes place via a protocol. The Socket provides a communication mechanism between two computers using TCP. A socket is one end-point of a two-way communication link between two programs running on the network. Socket classes are used to represent the connection between a client program and a server program as shown in Figure 2. There are two kinds of TCP sockets in Java.

- ✓ **ServerSocket** class is for servers.
- ✓ **Socket** class is for the client.

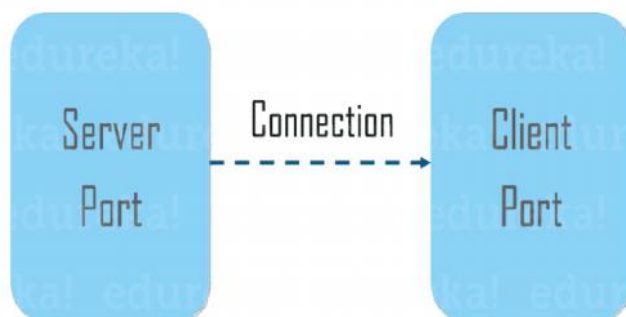


Figure 2: Client/server Connection

A **socket** is bound to a port number so that the TCP layer can identify the application that data is destined to be sent to. An endpoint is a combination of an IP address and a port number. The package in the Java platform provides a class, **Socket** that implements one side of a two-way connection between your Java program and another program on the network. The class sits on top of a platform-dependent implementation, hiding the details of any particular system from your Java program. By using the class instead of relying on native code, your Java programs can communicate over the network in a platform-independent fashion.

Socket Class

The `java.net.Socket` class allows us to create socket objects that help us in implementing all fundamental socket operations. We can perform various networking operations such as sending, reading data, and closing connections. Each Socket object that has been created using with `java.net.Socket` class has been associated exactly with 1 remote host. For connecting to another different host, we must create a new socket object. The syntax for importing Socket class from `java.net` package :

import java.net.Socket;

The following are the methods used in socket class:

- `public InputStream getInputStream()` : returns the `InputStream` attached with this socket.
- `public OutputStream getOutputStream()` : returns the `OutputStream` attached with this socket.
- `public synchronized void close()` : closes this socket

Methods used in Socket class

- `bind(SocketAddress bindpoint)`: This method is used for binding a socket to a local address.
- `Close()`: This method is used for terminating a socket.
- `connect(SocketAddress endpoint)`: This method is used in connecting a socket to the server.
- `getInetAddress()`: This method returns the address to which the socket is connected.
- `getInputStream()` : This method returns the input stream for the socket.
- `getKeepAlive()`: This method is used to check if `SO_KEEPALIVE` is enabled or not.
- `getLocalAddress()`: This method is used to fetch the local address to which the socket is bound.
- `getLocalPort()`: This method returns the local address to which the socket is bound.
- `getLocalSocketAddress()` : This method returns the endpoint for which the socket is bound.
- `getOutputStream()` : This method returns the output stream for the socket.
- `getPort()`: This method returns the remote port number with which the socket is associated to.
- `isBound()`: This method returns the binding state of the socket
- `isClosed()`: This method returns the closed state of the socket.
- `isConnected()` : This method returns the connection state of the socket.

Client-Side Programming

In the case of client-side programming, The client will first wait for the server to start. Once the server is up and running, it will send the requests to the server. After that, the client will wait for the response from the server. So, this is the whole logic of client and server communication.

Initiate a Client request

To initiate a client's request, you need to follow the below-mentioned steps:

- **Establish a Connection:** The very first step is to establish a socket connection. A socket connection implies that the two machines have information about each other's network location (IP Address) and TCP port.

You can create a Socket with the help of the below statement:

Socket socket = new Socket("127.0.0.1", 5000)

Here, the first argument represents the **IP address of the Server**. The second argument represents the **TCP Port**. (It is a number that represents which application should run on a server.)

- ✓ Here, the first argument represents the **IP address of the Server**.
- ✓ The second argument represents the **TCP Port**. (It is a number that represents which application should run on a server.)
- **Communication:** To communicate over a socket connection, streams are used for both input and output the data. After establishing a connection and sending the requests, you need to close the connection.
- **Closing the connection:** The socket connection is closed explicitly once the message to the server is sent.

Applications of Socket Class

- The Socket class is implemented in creating a stream socket and which is connected to a specified port number and port address.

public Socket(InetAddress address, int port)

- The Socket class is used for the creation of a socket and connecting to the specified remote address on the specified remote port in javax.net.

SocketFactory.createSocket(InetAddress address, int port, InetAddress localAddress, int localPort)

- In javax.ssl.net, the Socket class is used to return a socket layered over an existing socket connected to the named host at a given port.

SSLConnectionFactory.createSocket(Socket s, String host, int port, boolean autoClose)

- In javax.rmi.ssl, Socket class is used for creating an SSL socket.

SslRMIClientSocketFactory.createSocket(String host, int port)

- In java.nio.channels, Socket class is used for retrieving a socket associated with its channel

SocketChannel.socket()

13.3 ServerSocket Class

ServerSocket class is used for providing system-independent implementation of the server-side of a client/server Socket Connection. The constructor for ServerSocket throws an exception if it can't listen on the specified port (for example, the port is already being used). The ServerSocket class can be used to create a server socket. This object is used to establish communication with the clients. It is widely used so the applications of java.net.ServerSocket class which is as follows:

- In java.nio channel, ServerSocket class is used for retrieving a ServerSocket associated with this channel.
- In java.rmi.Server, ServerSocket class is used to create a server socket on the specified port (port 0 indicates an anonymous port).
- In javax.net, the ServerSocket is used widely to:
 - ✓ return an unbound server socket.
 - ✓ return a server socket bound to the specified port.
 - ✓ return a server socket bound to the specified port, and uses the specified connection backlog.
 - ✓ return a server socket bound to the specified port, with a specified listen backlog and local IP.

Server-Side Program

Server-side, a two-socket is required for better communication with the client. When a client opens a new Socket, the first socket is merely used to wait for the client's request (). And there's a simple old socket for communicating with the client. The remainder of the steps are the same, but we utilized the `getOutputStream()` method on the socket side to send the output.

Methods used in ServerSocket Class

`accept()`: Listens for a connection to be made to this socket and accepts it.

`Bind(SocketAddress endpoint)`: Binds the ServerSocket to a specific address (IP address and port number).

`getChannel()`: Returns the unique ServerSocketChannel object associated with this socket, if any.

`close()`: Closes this socket.

`getInetAddress()`: Returns the local address of this server socket.

`getLocalPort()`: Returns the port number on which this socket is listening.

`getLocalSocketAddress()`: Returns the address of the endpoint this socket is bound to, or null if it is not bound yet.

`getReuseAddress()`: Tests if `SO_REUSEADDR` is enabled.

`getReceiveBufferSize()`: Gets the value of the `SO_RCVBUF` option for this ServerSocket, that is the proposed buffer size that will be used for Sockets accepted from this ServerSocket.

`getSoTimeout()`: Retrieve setting for `SO_TIMEOUT`.

`implAccept(Socket s)`: Subclasses of ServerSocket use this method to override `accept()` to return their own subclass of the socket.

`setSocketFactory(SocketImplFactory fac)`: Sets the server socket implementation factory for the application.

**Example**

```
// SimpleServer.java: A simple server program.
import java.net.*;
import java.io.*;

public class SimpleServer {
    public static void main(String args[]) throws IOException {
        // Register service on port 1254
        ServerSocket s = new ServerSocket(1254);
        Socket s1=s.accept(); // Wait and accept a connection
        // Get a communication stream associated with the socket
        OutputStream s1out = s1.getOutputStream();
        DataOutputStream dos = new DataOutputStream (s1out);
        // Send a string!
        dos.writeUTF("Message from Server Side");
        // Close the connection, but not the server socket
        dos.close();
        s1out.close();
        s1.close();
    }
}
```

```

}
}

```



Example

// SimpleClient.java: A simple client program.

```

import java.net.*;
import java.io.*;

public class SimpleClient {

    public static void main(String args[]) throws IOException {
        // Open your connection to a server, at port 1254
        Socket s1 = new Socket("localhost",1254);
        // Get an input file handle from the socket and read the input
        InputStream s1In = s1.getInputStream();
        DataInputStream dis = new DataInputStream(s1In);
        String st = new String (dis.readUTF());
        System.out.println(st);
        // When done, just close the connection and exit
        dis.close();
        s1In.close();
        s1.close();
    }
}

```

Output

```

PS C:\Users\Harjinder Kaur\OneDrive\Documents\cap615\programs> jav
ac SimpleServer.java
PS C:\Users\Harjinder Kaur\OneDrive\Documents\cap615\programs> jav
a SimpleServer
PS C:\Users\Harjinder Kaur\OneDrive\Documents\cap615\programs>
PS C:\Users\Harjinder Kaur\OneDrive\Documents\cap615\programs> jav
ac SimpleClient.java
PS C:\Users\Harjinder Kaur\OneDrive\Documents\cap615\programs> jav
a SimpleClient
Hi client you are welcome
PS C:\Users\Harjinder Kaur\OneDrive\Documents\cap615\programs>

```

Figure 3: Output of Socket and ServerSocket Class

To get the output you have to open two command prompts simultaneously. First, you have to execute the ServerSocket class then Socket class is only the connection is established between client and server and you will get the desired output.

13.4 URL class

Java URL (Uniform Resource Locator) Class present in java.net package, deals with URL which uniquely identify or locate resources on internet. The following are the important Methods of URL class:

- ✓ **getProtocol()** : Returns protocol of URL
- ✓ **getHost()** : Returns hostname(domain name) of URL

- ✓ **getPort()** : Returns port number of URL
- ✓ **getFile()** : Returns filename of URL

Components of Java URL Class

The URL is made up of three or four parts; many URLs, are made up of three parts.



Example: <https://www.google.com/what-is-java>

Protocol – The protocol subpart of the URL specifies the protocol name, which in this case is HTTPS.

Hostname - The hostname, also known as the IP address or server name, is the name of the computer or server, as in www.google.com.

Port Number - The URL's port number is an optional component that gives the logical address of any resource that communicates over the Internet. Because we don't have the port number, each resource is assigned a 16-bit integer port number. It returns -1 if the port number is not present.

File Name – The pathname to a resource or file on the server is specified by the file name, which in this case is "what-is-java."

Constructors of Java URL Class

URL(String url) – This function returns the string url into a URL object.

URL(String protocol, String host, String file) – Creates a URL object from the protocol, host, and file supplied.

URL(String protocol, String host, int portno, String file) – Creates a URL object using the supplied protocol name, hostname, port number, and file name.

URL(String protocol, String host, int portno, String file, URLStreamHandler handler) – This function constructs a URL object using the protocol, host, port number, file, and handler parameters.

URL(URL context, String url) – Creates a URL object from the specified URL in the supplied context with this function.

URL(URL context, String url, URLStreamHandler handler) – This function produces a URL object with the supplied handler from the specified URL in the specified context.



Example

// Java program to demonstrate working of URL

```
import java.net.MalformedURLException;
```

```
import java.net.URL;
```

```
public class URLExample
```

```
{
```

```
    public static void main(String[] args) throws MalformedURLException
```

```
    {
```

```
        // creates a URL with string representation.
```

```
        URL url1 = new URL("https://ums.lpu.in/lpuums/home");
```

```
        // print the string representation of the URL.
```

```
        System.out.println(url1.toString());
```

```
        System.out.println();
```

```
        System.out.println("Different components of the URL");
```

```

        System.out.println("Protocol:- " + url1.getProtocol());
        System.out.println("Hostname:- " + url1.getHost());
        System.out.println("Default port:- " + url1.getDefaultPort());
        System.out.println("Query:- " + url1.getQuery());
        System.out.println("Path:- " + url1.getPath());
        System.out.println("File:- " + url1.getFile());
        System.out.println("Reference:- " + url1.getRef());
    }
}

```

Output

PS C:\Users\Harjinder Kaur\OneDrive\Documents\cap615\programs>javac URLExample.java

PS C:\Users\Harjinder Kaur\OneDrive\Documents\cap615\programs> java URLExample

https://ums.lpu.in/lpuums/home

Different components of the URL

Protocol:- https

Hostname:- ums.lpu.in

Default port:- 443

Query:- null

Path:- /lpuums/home

File:- /lpuums/home

Reference:- null

Summary

- Java Networking is a notion of combining two or more computing devices to share resources.
- Socket classes are used to represent the connection between a client program and a server program.
- ServerSocket class is used for providing system-independent implementation of the server-side of a client/server Socket Connection.
- The address of a resource on the World Wide Web is specified by the URL. The URL is a one-of-a-kind way to access things on the Internet.
- The java.net package provides two classes--Socket and ServerSocket--that implement the client-side of the connection and the server-side of the connection, respectively.
- When the client creates its socket object, it specifies the address of the server process, namely, the IP address of the server and the port number of the process.

Keywords

Hostname: The name of the machine or server is specified by the hostname, IP address, or server name of the URL.

TCP: The Transmission Control System (TCP) is a widely used data transmission protocol that supports client/server endpoints on a network.

Programming in JAVA

UDP: The User Datagram Mechanism (UDP) is a connectionless protocol for sending data packets between applications.

Connection-oriented sockets: They operate as a telephone: they must establish a connection and then hang up. Everything that flows between these two events arrives in the same order it was sent.

Connectionless sockets: It operates like the mail: delivery is not guaranteed, and multiple pieces of mail may arrive in a different order than they were sent.

Port Number: The URL's port number is an optional component that gives the logical address of any resource that communicates over the Internet

Self Assessment

1. Which steps occur when establishing a TCP connection between two computers using sockets?
 - A. The server instantiates a ServerSocket object, denoting which port number communication is to occur on
 - B. The server invokes the accept() method of the ServerSocket class. This method waits until a client connects to the server on the given port
 - C. After the server is waiting, a client instantiates a Socket object, specifying the server name and port number to connect to
 - D. All of the above

2. Which of the following is a stateless protocol?
 - A. HTML
 - B. XHTML
 - C. HTTP
 - D. All of the mentioned

3. Which of the following method returns the closed state of the socket.
 - A. closed()
 - B. isBound()
 - C. isClosed()
 - D. None of the above

4. Which of the following are not socket property?
 - A. onopen
 - B. readyState
 - C. onmessage
 - D. ready

5. _____method returns the endpoint for which the socket is bound.
 - A. getLocalAddress()
 - B. getLocalPort()
 - C. getLocalSocketAddress()
 - D. None

6. Which of the following is/are networking terminologies?
- A. IP Address
 - B. Protocol
 - C. MAC Address
 - D. All mentioned above
7. Which class can be used to create a server socket. This object is used to establish communication with the clients?
- A. ServerSocket
 - B. Socket
 - C. Both A & B
 - D. None of the above
8. Java program communicates over the network at the _____.
- A. Physical Layer
 - B. Application Layer
 - C. Data Link Layer
 - D. All of the above
9. Which of these is a protocol for breaking and sending packets to an address across a network?
- A. TCP/IP
 - B. DNS
 - C. Socket
 - D. Proxy Server
10. _____ is the Machine that places the request to access the data.
- A. Server Machine
 - B. Request Machine
 - C. Client Machine
 - D. None of these
11. Which of the following comes under important classes of Networking
- A. InetAddress
 - B. Socket and ServerSocket Class
 - C. URL class
 - D. All of the above
12. URL is an acronym for?
- A. Uniform Resource Locator
 - B. Unified Resource Locator
 - C. Uniform Restore Locator
 - D. Unified Restore Locator

13. _____ returns the port number on which this socket is listening.
- A. getLocalPort()
 - B. getPort()
 - C. getRemotePort()
 - D. None
14. A ServerSocket can connect to _____ clients.
- A. One
 - B. Zero
 - C. Multiple
 - D. None
15. Name the class which is used to create a port where the server will listen?
- A. Socket
 - B. ServerSocket
 - C. SocketServer
 - D. Datagram Socket

Answers for Self Assessment

- | | | | | |
|-------|-------|-------|-------|-------|
| 1. D | 2. C | 3. C | 4. D | 5. C |
| 6. D | 7. A | 8. B | 9. A | 10. C |
| 11. D | 12. A | 13. A | 14. C | 15. B |

Review Questions

1. "Network programming refers to writing programs that execute across multiple devices, in which the devices are connected via a network." Justify the statement with an example.
2. Discuss the significance of using the URL class. Explain the various components and methods of URL class by giving an example of each.
3. What is the importance of Socket class? Write a program for the implementation of the Socket class.
4. "ServerSocket Class is used for providing system-independent implementation of the server-side of a client/server Socket Connection." Comment on the statement with the help of an appropriate example.
5. Diagrammatically elaborate on the concept of networking. Elucidate the various networking terminologies used in java.
6. Differentiate connection-oriented and connection-less sockets by giving the example of each type.



Further Readings

Graba, J. (2006). An introduction to network programming with Java (pp. I-XI). Springer.

Harold, E. R. (2004). Java network programming. " O'Reilly Media, Inc."

Sawant, A. A., &Meshram, B. (2013). Network programming in Java using Socket. Google Scholar.

Reese, R. M. (2015). Learning Network Programming with Java. Packt Publishing Ltd.

Calvert, K. L., &Donahoo, M. J. (2011). TCP/IP sockets in Java: a practical guide for programmers. Morgan Kaufmann.

Harold, E. R. (2004). Java network programming. " O'Reilly Media, Inc."



Web Links

<https://docs.oracle.com/javase/tutorial/networking/sockets/index.html>

<https://www.edureka.co/blog/java-networking/>

<https://www.geeksforgeeks.org/socket-programming-in-java/>

<https://www.section.io/engineering-education/socket-programming-in-java/>

<https://www.baeldung.com/a-guide-to-java-sockets>

Unit 14: More on Network Programming

CONTENTS

Objectives

Introduction

14.1 URL

14.2 URL Class

14.3 URLConnection Class

14.4 DatagramSocket class

14.5 DatagramPacket Class

14.6 Socket Programming

Summary

Keywords

Self Assessment

Answers for Self Assessment

Further Readings

Objectives

After this unit you will be able to:

- Understand the concept, methods, and implementation of URL and URLConnection Class.
- Learn the basic concept DatagramSocket Class.
- Know the various constructors, methods, and implementation of DatagramSocket Class.
- Learn the basic concept of Socket programming.
- Understand the process of creating client and server.
- Implementation of Socket Programming.

Introduction

Networking is the process of sharing information and data between computing devices such as laptops, desktops, servers, smartphones, and tablets, as well as an ever-expanding array of IoT devices such as cameras, door locks, doorbells, refrigerators, audio/visual systems, thermostats, and various sensors.

The term "network programming" refers to the creation of programs that run on several devices (computers), all of which are connected via a network. A socket is one of the two ends of a two-way communication link between two networked programs. On its end, a client application builds a socket and attempts to connect it to a server. The server produces a socket object on its end when the connection is established. Writing to and reading from the socket allows the client and server to communicate.

14.1 URL

As many of you must be knowing that Uniform Resource Locator-URL is a string of text that identifies all the resources on the Internet, telling us the address of the resource, how to communicate with it, and retrieve something from it.

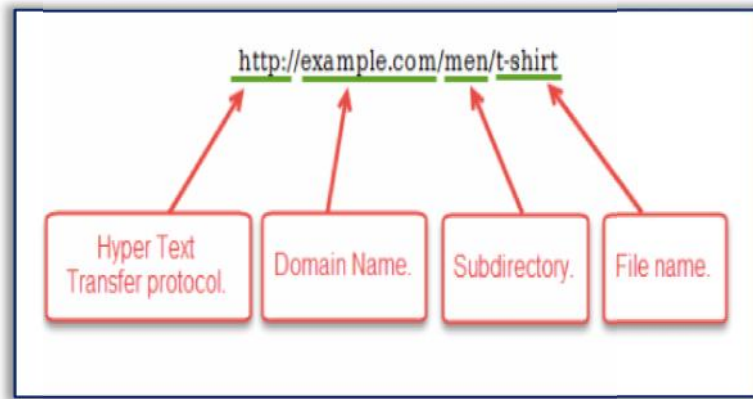


Figure 1: Parts of URL

A Simple URL looks like this:

1. **Protocol:** HTTP is the protocol here
2. **Hostname:** Name of the machine on which the resource lives.
3. **File Name:** The pathname to the file on the machine.
4. **Port Number:** Port number to which to connect (typically optional).



Note: You might think of a URL like a regular postal mail address: the scheme represents the postal service you want to use, the domain name is the city or town, and the port is like the zip code; the path represents the building where your mail should be delivered; the parameters represent extra information such as the number of the apartment in the building; and, finally, the anchor represents the actual person to whom you've addressed your mail.

14.2 URL Class

The URL class is the gateway to any of the resources available on the internet. A Class URL represents a Uniform Resource Locator, which is a pointer to a “resource” on the World Wide Web. A resource can point to a simple file or directory, or it can refer to a more complicated object, such as a query to a database or a search engine. The URL class in java is built in the `java.net.URL` package of java.

Constructors of the URL class

- **URL(String address) throws MalformedURLException:** It creates a URL object from the specified String.
- **URL(String protocol, String host, String file):** Creates a URL object from the specified protocol, host, and file name.
- **URL(String protocol, String host, int port, String file)** Creates a URL object from protocol, host, port, and file name.
- **URL(URL context, String spec):** Creates a URL object by parsing the given spec in the given context.
- **URL(String protocol, String host, int port, String file, URLStreamHandler handler)** Creates a URL object from the specified protocol, host, port number, file, and handler.
- **URL(URL context, String spec, URLStreamHandler handler)** Creates a URL by parsing the given spec with the specified handler within a specified context.

Methods of URL Class

- **public String getProtocol()** – This function gives the protocol used by the URL.
- **public String getHost()** – This function gives the hostname used by the URL.
- **public String getPort()** – This function gives the Port Number used by the URL.
- **public String getFile()** – This function gives the file name.
- **public String getAuthority()** – This function gives the authority of an URL if empty returns null.
- **public String toString()** – This function gives the URL representation in string.
- **public String getQuery()** – This function gives the query of the URL. Query part in URL is present after '?'.
 - **public String getDefaultPort()** – This function gives the URL default port.
 - **public String getPath()** – This function gives the path of the URL if empty returns null.
 - **public boolean equals(Object obj)** – This function compares the two URL object by the given URL object.
 - **public Object getContent()** – This function gives the URL content.
 - **public String getRef()** – This function gives the reference of the URL. The reference part in the URL is present after '#'.



Example

// Java program to demonstrate the working of URL

```
import java.net.MalformedURLException;
```

```
import java.net.URL;
```

```
public class URLExample
```

```
{
    public static void main(String[] args) throws MalformedURLException
    {
        // creates a URL with string representation.
        URL url1 = new URL("https://ums.lpu.in/lpuums/home");
        // print the string representation of the URL.
        System.out.println(url1.toString());
        System.out.println();
        System.out.println("Different components of the URL");
        System.out.println("Protocol:- " + url1.getProtocol());
        System.out.println("Hostname:- " + url1.getHost());
        System.out.println("Default port:- " + url1.getDefaultPort());
        System.out.println("Query:- " + url1.getQuery());
        System.out.println("Path:- " + url1.getPath());
        System.out.println("File:- " + url1.getFile());
        System.out.println("Reference:- " + url1.getRef());
    }
}
```

Output

```
PS C:\Users\Harjinder Kaur\OneDrive\Documents\cap615\programs>javac URLExample.java
```

```
PS C:\Users\Harjinder Kaur\OneDrive\Documents\cap615\programs> java URLExample
```

```
https://ums.lpu.in/lpuums/home
```

Different components of the URL

Protocol:- https

Hostname:- ums.lpu.in

Default port:- 443

Query:- null

Path:- /lpuums/home

File:- /lpuums/home

Reference:- null

14.3 URLConnection Class

The URLConnection class represents a communication link between the URL and the application. It can be used to read and write data to the specified resource referred by the URL. There are mainly two subclasses that extend the URLConnection class-

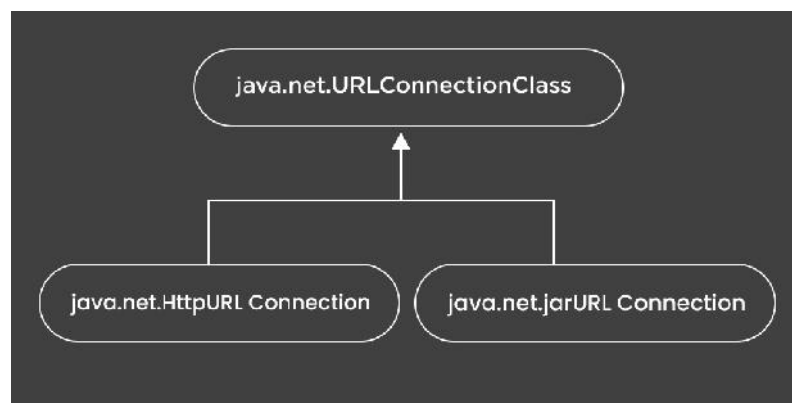


Figure 2: Types of URLConnection

- ✓ **HttpURLConnection:** If we are connecting to any url which uses “http” as its protocol, then HttpURLConnection class is used.
- ✓ **JarURLConnection:** If however, we are trying to establish a connection to a jar file on the web, then JarURLConnection is used.

Features of URLConnection class

- URLConnection is an abstract class. The two subclasses HttpURLConnection and JarURLConnection make the connection between the client Java program and URL resource on the internet.
- With the help of URLConnection class, a user can read and write to and from any resource referenced by an URL object.
- Once a connection is established and the Java program has an URLConnection object, we can use it to read or write or get further information like content length, etc.

Methods of URLConnection Class

- **URLConnection.openConnection()** : opens the connection to the specified URL.

- **Object getContent()** : retrieves the content of the URLConnection.
- **Map<String,List>getHeaderFields()** : returns the map containing the values of various header fields in the http header.
- **getContentEncoding()** : Returns the value of the content-encoding header field.
- **getContentLength()** : It returns the length of the content header field.
- **getDate()** : It returns value of date in header field.
- **getHeaderField(int i)** : This method returns the value of ith index of header.
- **getHeaderField(String field)** : It returns the value of field named "field" in header
- **OutputStreamgetOutputStream()** : It returns the output stream to this connection.
- **InputStreamgetInputStream()** : Method returns the input stream to this open connection.
- **setAllowUserInteraction(boolean)** : Setting this true means a user can interact with the page. The default value is true.
- **setDefaultUseCaches(boolean)** : Sets the default value of useCache field as the given value.
- **setDoInput(boolean)**: This method sets if the user is allowed to take input or not.
- **setDoOutput(boolean)**: It sets if the user is allowed to write on the page. The default value is false since most of the url doesn't allow to write.



Example

// Implementation of URLConnection Class

```
import java.io.*;
import java.net.*;

public class URLConnectionclass {
    public static void main(String[] args){
        try{
            URL url=new URL("http://mail.google.com/mail/u/0/?ogbl#inbox");
            URLConnectionurlcon=url.openConnection();
            InputStream stream=urlcon.getInputStream();
            int i;
            while((i=stream.read())!=-1){
                System.out.print((char)i);
            }
        }catch(Exception e){System.out.println(e);}
    }
}
```

Output

PS C:\Users\Harjinder Kaur\OneDrive\Documents\cap615>javac URLConnectionclass.java

PS C:\Users\Harjinder Kaur\OneDrive\Documents\cap615> java URLConnectionclass

<HTML>

<HEAD>

<TITLE>Moved Permanently</TITLE>

```

</HEAD>
<BODY BGCOLOR="#FFFFFF" TEXT="#000000">
<H1>Moved Permanently</H1>
The document has moved <A HREF="https://mail.google.com/mail/u/0/">here</A>.
</BODY>
</HTML>

```

14.4 DatagramSocket class

Java DatagramSocket and DatagramPacket classes are used for connection-less socket programming using the UDP instead of TCP. Datagrams are a collection of information sent from one device to another device via the established network. When the datagram is sent to the targeted device, there is no assurance that it will reach the target device safely and completely. . It may get damaged or lost in between. Likewise, the receiving device also never knows if the datagram received is damaged or not. The UDP protocol is used to implement the datagrams in Java.

DatagramSocket class represents a connection-less socket for sending and receiving datagram packets. It is a mechanism used for transmitting datagram packets over the network. A datagram is a piece of information but there is no guarantee of its content, arrival, or arrival time.

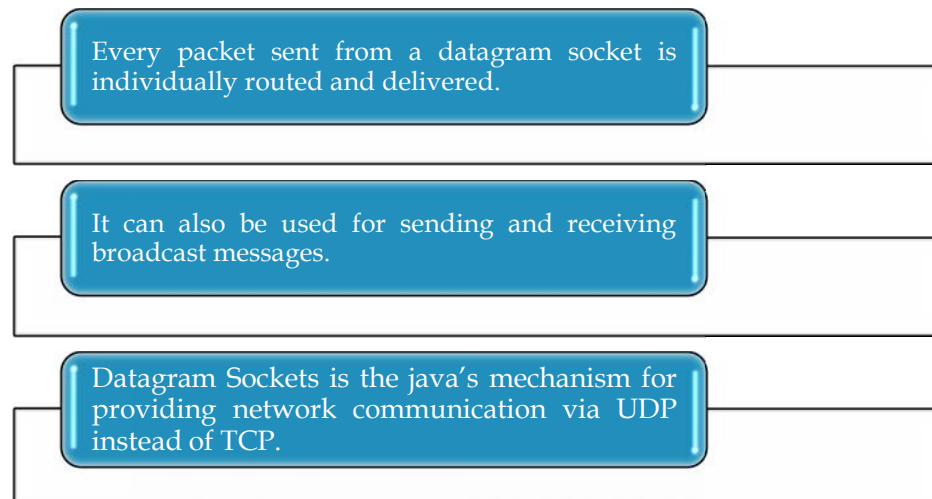


Figure 3: Features of DatagramSocket

Constructors of DatagramSocket Class

- **DatagramSocket():** Creates a datagramSocket and binds it to any available port on a local machine. If this constructor is used, the OS would assign any port to this socket.
- **DatagramSocket(DatagramSocketImpl)** : Creates an unbound datagram socket with given datagramImpl.
- **DatagramSocket(int port):** Creates and binds the datagram socket to the specified port. The socket will be bound to the wildcard address chosen by the kernel.
- **DatagramSocket(int port, InetAddressladdr):** Constructs a datagram socket and binds it to a specified port and initaddress.
- **DatagramSocket(SocketAddressbindaddr):** Constructs a new socket object and binds it to the specified socket address(IP address+port number).

Methods of DatagramSocket class

- **bind():** Binds this socket to the specified address and port number.

- **connect():** Connects to the specified address and port. After connecting to the remote host, this socket can send or receive a packet from this remote host only.
- **disconnect():** Disconnects the socket. If the socket is not connected, then this method has no effect.
- **isClosed():** Returns the boolean value indicating if the socket is closed or not.
- **getChannel():** Returns a data channel if any associated with this socket.
- **close():** Closes this datagram socket. Any pending receive call will throw `SocketException`.
- **setBroadcast() :** Sets the value of `SO_BROADCAST` socket option.
- **getBroadcast():** Returns true if broadcast is enabled, false otherwise.
- **setReuseAddress():** Sometimes it may be necessary to bind multiple sockets to the same address. Enabling this option enables another socket to bind to the same address as this. It must be set before a call to `bind()` is made. It sets the value of the `SO_REUSEADDR` socket option.
- **setReceiveBufferSize():** Used to set a limit to the maximum size of the packet that is received on this socket.
- **getReceiveBufferSize():** Returns the value of `SO_RCVBUF` option of this socket.
- **setSoTimeout():** This is used to set the waiting time for receiving a datagram packet. As a call to `receive()` method blocks execution of the program indefinitely until a packet is received, this method can be used to limit that time.
- **getSoTimeout():** This is used to set the waiting time for receiving a datagram packet. As a call to `receive()` method blocks execution of the program indefinitely until a packet is received, this method can be used to limit that time.
- **getLocalPort():** Returns the port on a local machine to which this socket is bound.
- **getLocalAddress():** Returns the local address to which this socket is bound.
- **receive():** It is used to receive the packet from a sender. When a packet is successfully received, the buffer of the packet is filled with the received message. The packet also contains valuable information like the sender's address and the port number. This method waits till a packet is received.
- **send():** Sends a datagram packet from this socket. It should be noted that the information about the data to be sent, the address to which it is sent, etc are all handled by the packet itself.

14.5 DatagramPacket Class

A message that can be delivered or received is known as a `DatagramPacket`. It's a container for data. It's possible that if you send many packets, they'll arrive in any order. Furthermore, packet delivery cannot be assured.

Constructors of `DatagramPacket` class

- **`DatagramPacket(byte[] barr, int length):`** it creates a datagram packet. This constructor is used to receive the packets.
- **`DatagramPacket(byte[] barr, int length, InetAddress address, int port):`** it creates a datagram packet. This constructor is used to send the packets.

Table 1: Methods of DatagramPacket Class

Method	Description
<code>InetAddress getAddress()</code>	It returns the IP address of the machine to which the datagram is being sent or from which the datagram was received.
<code>byte[] getData()</code>	It returns the data buffer.
<code>int getLength()</code>	It returns the length of the data to be sent or the length of the data received.
<code>int getOffset()</code>	It returns the offset of the data to be sent or the offset of the data received.
<code>int getPort()</code>	It returns the port number on the remote host to which the datagram is being sent or from which the datagram was received.
<code>SocketAddress getSocketAddress()</code>	It gets the SocketAddress (IP address + port number) of the remote host that the packet is being sent to or is coming from.
<code>void setAddress(InetAddress addr)</code>	It sets the IP address of the machine to which the datagram is being sent.
<code>void setData(byte[] buff)</code>	It sets the data buffer for the packet.
<code>void setLength(int length)</code>	It sets the length of the packet.
<code>void setPort(int iport)</code>	It sets the port number on the remote host to which the datagram is being sent.
<code>void setSocketAddress(SocketAddress addr)</code>	It sets the SocketAddress (IP address + port number) of the remote host to which the datagram is being sent.

**Example**

```
//Code for DSender:
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
public class DSender
{
    public static void main(String[] args) throws Exception
    {
        DatagramSocket ds = new DatagramSocket();
        String str = "Implementation of Datagram Socket";
        InetAddress ia = InetAddress.getByName("127.0.0.1");
        DatagramPacket dp = new DatagramPacket(str.getBytes(), str.length(), ia, 3000);
        ds.send(dp);
    }
}
```

```
ds.close();
}
}
```



Example

//Code for DReceiver:

```
import java.net.DatagramPacket;
import java.net.DatagramSocket;
public class DReceiver
{
    public static void main(String[] args) throws Exception
    {
        DatagramSocket ds = new DatagramSocket(3000);
        byte[] buf = new byte[1024];
        DatagramPacket dp = new DatagramPacket(buf, 1024);
        ds.receive(dp);
        String strRecv = new String(dp.getData(), 0, dp.getLength());
        System.out.println(strRecv);
        ds.close();
    }
}
```

Output

```
PS C:\Users\Harjinder Kaur\OneDrive\Documents\cap615\programs>javac DReceiver.java
```

```
PS C:\Users\Harjinder Kaur\OneDrive\Documents\cap615\programs> java DReceiver
Implementation of Datagram Socket
```

```
PS C:\Users\Harjinder Kaur\OneDrive\Documents\cap615\programs>javac DSender.java
```

```
PS C:\Users\Harjinder Kaur\OneDrive\Documents\cap615\programs> java DSender
```

14.6 Socket Programming

Java Socket programming is used for communication between the applications running on different JRE. It can be connection-oriented or connection-less. Socket and ServerSocket classes are used for connection-oriented socket programming. DatagramSocket and DatagramPacket classes are used for connection-less socket programming. The client in socket programming must know two information:

- IP Address of Server
- Port number.

Creating Server

To create the server application, we need to create the instance of ServerSocket class. Here, we are using the 9999 port number for the communication between the client and server. You may also choose any other port number. The accept() method waits for the client. If the client connects with the given port number, it returns an instance of Socket.

- ✓ ServerSocket ss=new ServerSocket(9999);

- ✓ Socket s=ss.accept(); //establishes connection and waits for the client

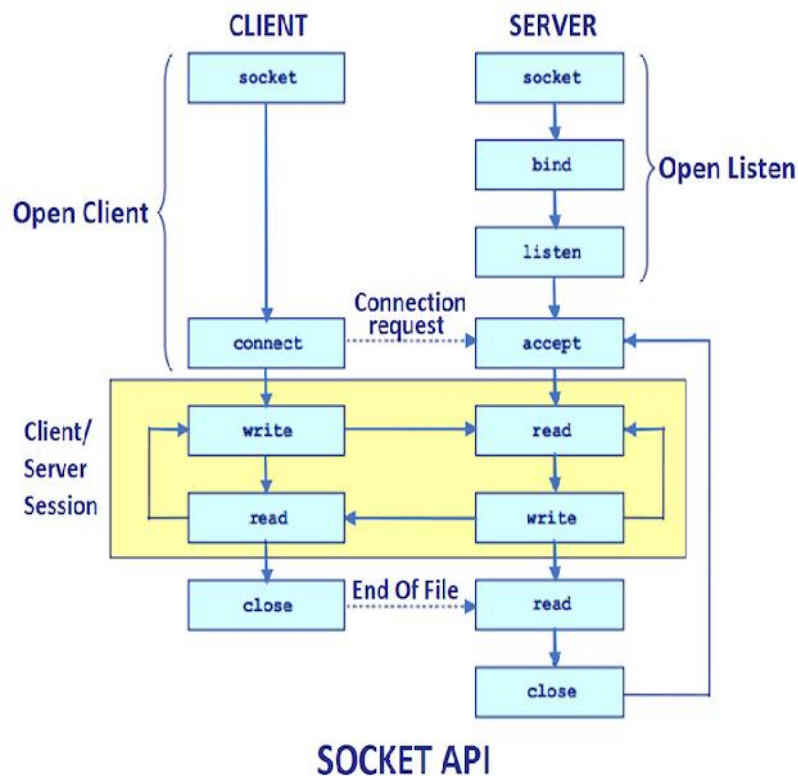


Figure 4: Client-server Commutation

Here, we are going to make one-way client and server communication. In this application, the client sends a message to the server, the server reads the message and prints it. Here, two classes are being used: Socket and ServerSocket. The Socket class is used to communicate client and server. Through this class, we can read and write a message. The ServerSocket class is used at the server-side. The accept() method of ServerSocket class blocks the console until the client is connected. After the successful connection of the client, it returns the instance of Socket at the server-side.



Caution: When developing a proprietary application, the developer must be careful not to use one of the well-known port numbers defined in the RFCs.

Creating Client

To create the client application, we need to create the instance of the Socket class. Here, we need to pass the IP address or hostname of the Server and a port number.

Socket s=new Socket("localhost",9999); Here, we are using "localhost" because our server is running on same system.



Example

```
// Implementation of socket client side
import java.net.*;

public class MyClient {

public static void main(String[] args) {

try{

Socket s=new Socket("localhost",6060);

DataOutputStreamdout=new DataOutputStream(s.getOutputStream());

dout.writeUTF("Connected to Server");
```

```
dout.flush();
dout.close();
s.close();
}catch(Exception e){System.out.println(e);}
}
}
```



Example

```
//Implementation of socket on server
import java.io.*;
import java.net.*;
public class MyServer {
public static void main(String[] args){
try{
ServerSocket ss=new ServerSocket(6060);
Socket s=ss.accept();//establishes connection
DataInputStream dis=new DataInputStream(s.getInputStream());
String str=(String)dis.readUTF();
System.out.println("message= "+str);
ss.close();
}catch(Exception e){System.out.println(e);}
}
}
```

Output

```
PS C:\Users\Harjinder Kaur\OneDrive\Documents\cap615\programs>javac MyServer.java
PS C:\Users\Harjinder Kaur\OneDrive\Documents\cap615\programs> java MyServer
PS C:\Users\Harjinder Kaur\OneDrive\Documents\cap615\programs>javac MyClient.java
PS C:\Users\Harjinder Kaur\OneDrive\Documents\cap615\programs> java MyClient
message= Connected to Server
```

Summary

- URLConnection is the superclass of all the classes that represent a communication link between the application and a URL.
- The **java.net.URL** class represents a URL and has a complete set of methods to manipulate URLs in Java.
- Socket programming is a means of communicating data between two computers across a network.
- Java DatagramSocket and DatagramPacket classes are used for connection-less socket programming using the UDP instead of TCP.

- The DatagramPacket classes and DatagramSocket are used for connection-less socket programming.
- A URL class represents a Uniform Resource Locator, which is a pointer to a “resource” on the World Wide Web.

Keywords

Socket: One endpoint of a two-way communication link between two programs that are running on a network.

TCP: Transmission Control Protocol (TCP) is a widely used protocol for data transmission on a network that supports client/server endpoints.

DatagramSocket: A datagram socket is the sending or receiving point for a packet delivery service. Each packet sent or received on a datagram socket is individually addressed and routed.

URL: URL is an acronym for *Uniform Resource Locator* and is a reference (an address) to a resource on the Internet.

DatagramPacket: DatagramPacket is a message that can be sent or received. If you send multiple packets, they may arrive in any order.

Self Assessment

1. Which of these exceptions is thrown by the URL class's constructors?
 - A. URLNotFound
 - B. URLSourceNotFound
 - C. MalformedURLException
 - D. URLNotFoundException
2. Which of these methods is used to know host of an URL?
 - A. host()
 - B. getHost()
 - C. GetHost()
 - D. gethost()
3. Which of these transfer protocols must be used so that the URL can be accessed by the URLConnection class object?
 - A. http
 - B. HTTPS
 - C. Any Protocol can be used
 - D. None of the mentioned
4. Which of these methods is used to know the type of content used in the URL?
 - A. ContentType()
 - B. contentType()
 - C. getContentType()
 - D. GetContentType()

5. Which method of URL class represents a URL and has the complete set of methods to manipulate URL in Java?
- A. java.net.URL
 - B. java.net.URLConnection
 - C. Both A & B
 - D. None of the above
6. The URLConnection class can be used to read and write data to the specified resource referred to by the URL
- A. True
 - B. False
 - C. Cant'say
 - D. May be
7. Which of the following is/are the methods of URL class?
- A. getAuthority()
 - B. toString()
 - C. getQuery()
 - D. All of the above.
8. Which of the following comes under the method of URLConnection Class?
- A. getLength()
 - B. getDate()
 - C. getHeaderField(int i)
 - D. All
9. Which of these package contains classes and interfaces for networking?
- A. java.io
 - B. java.util
 - C. java.net
 - D. java.network
10. To create the server application, we need to create the instance of _____ class.
- A. ServerSocket
 - B. Socket
 - C. Server
 - D. ServerReader
11. What does the java.net.InetAddress class represent?
- A. Socket
 - B. IP Address
 - C. Protocol
 - D. MAC Address

12. CP,FTP,Telnet,SMTP,POP etc. are examples of ?

- A. Socket
- B. IP Address
- C. Protocol
- D. MAC Address

13. Which class represents an Internet Protocol address?

- A. InetAddress
- B. Address
- C. IP Address
- D. TCP Address

14. Which constructor of the Datagram Socket class is used to create a datagram socket and binds it with the given Port Number?

- A. Datagram Socket(int port)
- B. Datagram Socket(int port, Int Address address)
- C. Datagram Socket()
- D. Datagram Socket(int address)

15. Which classes are used for connection-less socket programming?

- A. Datagram Socket
- B. Datagram Packet
- C. Both Datagram Socket & Datagram Packet
- D. Server Socket

Answers for Self Assessment

- | | | | | |
|-------|-------|-------|-------|-------|
| 1. C | 2. B | 3. A | 4. C | 5. A |
| 6. A | 7. A | 8. D | 9. C | 10. A |
| 11. B | 12. C | 13. A | 14. B | 15. C |

Review Questions

1. “The URL class is the gateway to any of the resources available on the internet.” Comment on the statement by explaining the different components of the URL.
2. Differentiate between DatagramSocket and DatagramPacket class. Write a program to implement the same.
3. “The URLConnection class represents a communication link between the URL and the application.” Justify the statement by giving the different types of URLConnection class along with its various methods.
4. What is the significance of socket programming. How we establish a connection between client and server using socket programming.

5. With the help of an appropriate example differentiate between URL and URLConnection class.
6. Explain the difference in working of TCP and UDP with the help of an example.



Further Readings

Graba, J. (2006). An introduction to network programming with Java (pp. I-XI). Springer.

Ciubotaru, B., & Muntean, G. M. (2013). Advanced Network Programming-Principles and Techniques: Network Application Programming with Java. Springer Science & Business Media.

Calvert, K. L., & Donahoo, M. J. (2011). TCP/IP sockets in Java: a practical guide for programmers. Morgan Kaufmann.

Calvert, K. L., & Donahoo, M. J. (2011). TCP/IP sockets in Java: a practical guide for programmers. Morgan Kaufmann.

Calvert, K. L., & Donahoo, M. J. (2011). TCP/IP sockets in Java. Morgan Kaufmann Publishers.

Donahoo, M. J., & Calvert, K. L. (2009). *TCP/IP sockets in C: a practical guide for programmers*. Morgan Kaufmann.



Web Links

<https://codezup.com/socket-programming-client-and-server-in-java-example/>

<https://www.javatpoint.com/socket-programming>

<https://www.section.io/engineering-education/socket-programming-in-java/>

<https://docs.oracle.com/javase/8/docs/api/java/net/URLConnection.html>

LOVELY PROFESSIONAL UNIVERSITY

Jalandhar-Delhi G.T. Road (NH-1)
Phagwara, Punjab (India)-144411
For Enquiry: +91-1824-521360
Fax.: +91-1824-506111
Email: odl@lpu.co.in

