CS-22 Programming with Java

BCA Semester – 4

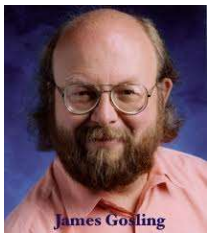
H & H B Kotak Institute of Science, Rajkot

BCA Department

Unit -1

History, Introduction and Language Basics, Classes and Objects

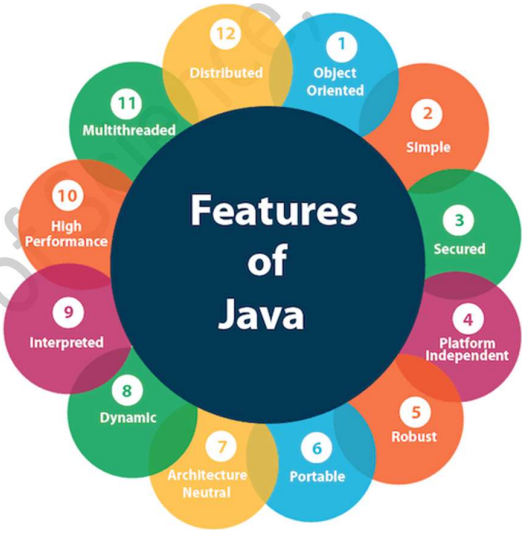
History and Features of Java



The history of Java is very interesting. Java was originally designed for interactive television, but it was too advanced technology for the digital cable television industry at the time. The history of Java starts with the Green Team. Java team members (also known as Green Team), initiated this project to develop a language for digital devices such as set-top boxes, televisions, etc. However, it was suited for internet programming. Later, Java technology was incorporated by Netscape.

The principles for creating Java programming were "Simple, Robust, Portable, Platform-independent, Secured, High Performance, Multithreaded, Architecture Neutral, Object-Oriented, Interpreted, and Dynamic". Java was developed by James Gosling, who is known as the father of Java, in 1995. James Gosling and his team members started the project in the early '90s. James Gosling - founder of java Currently, Java is used in internet programming, mobile devices, games, e-business solutions, etc. There are given significant points that describe the history of Java.

Features of JAVA



Simple: Java is very easy to learn, and its syntax is simple, clean and easy to understand. According to Sun, Java language is a simple programming language because: Java syntax is based on C++ (so easier for programmers to learn it after C++). Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc. There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java.

- **Object-Oriented:** Java is an object-oriented programming language. Everything in Java is an object. Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behavior. Object-oriented programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules.
- **Portable:** Portable – Being architecture-neutral and having no implementation dependent aspects of the specification makes Java portable. Compiler in Java is written in ANSI C with a clean portability boundary, which is a POSIX subset.
- **Platform Independent:** Java is platform independent because it is different from other languages like C, C++, etc. which are compiled into platform specific machines while Java is a write once, run anywhere language. A platform is the hardware or software environment in which a program runs. There are two types of platforms software-based and hardware-based. Java provides a software-based platform. The Java platform differs from most other platforms in the sense that it is a software-based platform that runs on the top of other hardware-based platforms. It has two components:
- **Secured:** Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because: No explicit pointer, and Java Programs run inside a virtual machine sandbox. Java language provides these securities by default. Some security can also be provided by an application developer explicitly through SSL, JAAS, Cryptography, etc.
    - **Bytecode Verifier:** It checks the code fragments for illegal code that can violate access right to objects.
    - **Security Manager:** It determines what resources a class can access such as reading and writing to the local disk.

- **Robust:** Robust simply means strong. Java is robust because It uses strong memory management, There is a lack of pointers that avoids security problems, and There is automatic garbage collection in java which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java

application anymore. There are exception handling and the type checking mechanism in Java. All these points make Java robust.

- **Architecture-neutral:** Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed. In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. However, it occupies 4 bytes of memory for both 32 and 64-bit architectures in Java.
- **High-performance:** Java is faster than other traditional interpreted programming languages because Java bytecode is "close" to native code. It is still a little bit slower than a compiled language (e.g., C++). Java is an interpreted language that is why it is slower than compiled languages, e.g., C, C++, etc.
- **Multi-threaded:** A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multi-media, Web applications, etc.
- **Distributed:** Java is distributed because it facilitates users to create distributed applications in Java. RMI and EJB are used for creating distributed applications. This feature of Java makes us able to access files by calling the methods from any machine on the internet.
- **Dynamic:** Java is a dynamic language. It supports dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages, i.e., C and C++.Java supports dynamic compilation and automatic memory management (garbage collection).

JDK, JVM and JRE

- **JVM:** JVM (Java Virtual Machine) is an abstract machine. It is called a virtual machine because it doesn't physically exist. It is a specification that provides a runtime environment in which Java bytecode can be executed. It can also run those programs which are written in other languages and compiled to Java bytecode. JVMs are available for many hardware and software platforms. JVM, JRE, and JDK are platform dependent because the configuration of each OS is different from each other. However, Java is platform independent. There are three notions of the JVM: specification, implementation, and instance. The JVM performs Loads code, Verifies code, Executes code, and Provides runtime environment.
- **JRE:** JRE is an acronym for Java Runtime Environment. It is also written as Java RTE. The Java Runtime Environment is a set of software tools which are used for developing Java applications. It is used to provide the runtime environment. It is the implementation of JVM. It physically exists. It contains a set of libraries + other files that JVM uses at runtime. The implementation of JVM is also actively released by other companies besides Sun Micro Systems.
- **JDK:** JDK is an acronym for Java Development Kit. The Java Development Kit (JDK) is a software development environment which is used to develop Java applications and applets. It physically exists. It contains JRE + development tools.JDK is an implementation of any one of the below given Java Platforms released by Oracle Corporation: Standard Edition Java Platform, Enterprise Edition Java Platform, and Micro Edition Java Platform.

JDK Tools

The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), etc. to complete the development of a Java Application.

Data Type

- **Integer:** 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). Its minimum value is - 2,147,483,648and maximum value is 2,147,483,647. Its default value is 0.The int data type is generally used as a default data type for integral values unless if there is no problem about memory.

  Example: int a = 100000, int b = -200000.
- **Float:** The float data type is a single-precision 32-bit IEEE 754 floating point.Its value range is unlimited. It is recommended to use a float (instead of double) if you need to save memory in large arrays of

floating point numbers. 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

- **Character:** 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'

- **Boolean:** The Boolean data type is used to store only two possible values: true and false. This data type is used for simple flags that track true/false conditions. The Boolean data type specifies one bit of information, but its "size" can't be defined precisely.

  Example: Boolean one = false

Java Tokens

Keywords: Java keywords are also known as reserved words. Keywords are particular words which acts as a key to a code. These are predefined words by Java so it cannot be used as a variable or object name. For Example .. abstract, byte, break, Boolean etc.

Binary Literal: Java added a new feature Binary Literal in Java 7. I allow you to express integral types (byte, short, int, and long) in binary number system. To specify a binary literal, add the prefix 0b or 0B to the integral value.

Identifier: A valid identifier in java…

- Must begin with a letter (A to Z or a to z), currency character ($) or an underscore (_).
- Can have any combination of characters after the first character.
- Cannot be a keyword.

Comments: The Java comments are the statements that are not executed by the compiler and interpreter. The comments can be used to provide information or explanation about the variable, method, class or any statement. It can also be used to hide program code. Types of Java Comments

There are three types of comments in Java.

1)Single Line Comment (//)

2)Multi Line Comment (/*….*/)

3)Documentation Comment (/**….*/)

Operators

Operator in Java is a symbol that is used to perform operations on variables and values. For example: +, -, *, / etc. There are many types of operators in Java which are given below:

Arithmetic Operator:

Operator	Description	Example
+ (Addition)	Adds values on either side of the operator.	A + B will give 30
- (Subtraction)	Subtracts right-hand operand from left-hand operand.	A - B will give -10
* (Multiplication)	Multiplies values on either side of the operator.	A * B will give 200
/ (Division)	Divides left-hand operand by right-hand operand.	B / A will give 2
% (Modulus)	Divides left-hand operand by right-hand operand and returns remainder.	B % A will give 0

++ (Increment)	Increases the value of operand by 1.	B++ gives 21
-- (Decrement)	Decreases the value of operand by 1.	B-- gives 19

Relational Operator:

Operator	Description	Example
== (equal to)	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!= (not equal to)	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
> (greater than)	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
< (less than)	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>= (greater than or equal to)	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<= (less than or equal to)	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Boolean Operator:

Operator	Description	Example
== (equal to)	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!= (not equal to)	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
> (greater than)	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
< (less than)	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>= (greater than or equal to)	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<= (less than or equal to)	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.
&& (logical and)	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false
|| (logical or)	Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.	(A || B) is true
! (logical not)	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true

Logical Operator:

Operator	Description	Example
&& (logical and)	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false
|| (logical or)	Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.	(A || B) is true
! (logical not)	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true

Bitwise Operator:

Operator	Description	Example
& (bitwise and)	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
| (bitwise or)	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101
^ (bitwise XOR)	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~ (bitwise compliment)	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<< (left shift)	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>> (right shift)	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 1111
>>> (zero fill right shift)	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.	A >>>2 will give 15 which is 0000 1111

Assignment Operator:

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand.	C = A + B will assign value of A + B into C
+=	Add AND assignment operator. It adds right operand to the left operand and assign the result to left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts right operand from the left operand and assign the result to left operand.	C -= A is equivalent to C = C – A

*=	Multiply AND assignment operator. It multiplies right operand with the left operand and assign the result to left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides left operand with the right operand and assign the result to left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assign the result to left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

Unary Operator:

Operator	Description	Example
~ (bitwise compliment)	Binary One's Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
++ (Increment)	Increases the value of operand by 1.	B++ gives 21
-- (Decrement)	Decreases the value of operand by 1.	B-- gives 19

Type Casting

A Type casting in Java is used to convert objects or variables of one type into another. When we are converting or assigning one data type to another they might not compatible. If it is suitable then it will do smoothly otherwise chances of data loss. Java Type Casting is classified into two types. Widening Casting (Implicit) – Automatic Type Conversion, and Narrowing Casting (Explicit) – Need Explicit Conversion.

Decision Statements

IF Statement: The Java if statement is used to test the condition. It checks Boolean condition: true or false. There are various types of if statement in Java like simple if, if…else, if…else if.. (Leader iif), and Nest if. The following is the simple syntax:

```
if(condition){
   //code to be executed
}
```

Switch Statement: The Java switch statement executes one statement from multiple conditions. It is like if-else-if ladder statement. The switch statement works with byte, short, int, long, enum types, String and some wrapper types like Byte, Short, Int, and Long. Since Java 7, you can use strings in the switch statement. In other words, the switch statement tests the equality of a variable against multiple values. The following is syntax:

```
switch(expression){
      case value1:
         //code to be executed;
         break;  //optional
      case value2:
         //code to be executed;
         break;  //optional
      ......
```

```
        default:
           code to be executed if all cases are not matched;
}
```

Looping Statements

For Loop: The Java for loop is used to iterate a part of the program several times. If the number of iterations is fixed, it is recommended to use for loop. The following is the syntax:

```
for (initialization; condition; update) {
      //statement or code to be executed
}
```

While Loop: The Java while loop is used to iterate a part of the program several times. If the number of iterations is not fixed, it is recommended to use while loop. The following is the syntax:

```
while(condition){
    //code to be executed
}
```

Do…While Loop: The Java do-while loop is used to iterate a part of the program several times. If the number of iterations is not fixed and you must have to execute the loop at least once, it is recommended to use do-while loop. The Java do-while loop is executed at least once because condition is checked after loop body. The following is the syntax:

```
do {
      //code to be executed
} while(condition);
```

Jumping Statements

- **Break:** When a break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop. The Java break statement is used to break loop or switch statement. It breaks the current flow of the program at specified condition. In case of inner loop, it breaks only inner loop. We can use Java break statement in all types of loops such as for loop, while loop and do-while loop. The Syntax:

  ```
  break;
  ```

- **Continue:** The continue statement is used in loop control structure when you need to jump to the next iteration of the loop immediately. It can be used with for loop or while loop. The Java continue statement is used to continue the loop. It continues the current flow of the program and skips the remaining code at the specified condition. In case of an inner loop, it continues the inner loop only. We can use Java continue statement in all types of loops such as for loop, while loop and do-while loop. The Syntax:

  ```
  continue;
  ```

- **Return:** Java return keyword is used to complete the execution of a method. The return followed by the appropriate value that is returned to the caller. This value depends on the method return type like int method always return an integer value. It is used to exit from the method. It is not allowed to use return keyword in void method. The value passed with return keyword must match with return type of the method.

Array

Array is a collection of elements which share same name and same data type. Array element can be accessed by index number. Index number starts from 0 to length – one. Array can be of different types, which describes as follows.

- **One Dimensioal Array:** A one-dimensional array is, essentially, a list of like-typed variables. To create an array, you first must create an array variable of the desired type. The general form of a one-dimensional array declaration is:

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

- **Multidimensional Arrays:** In Java, multidimensional arrays are actually array of arrays. These, as you might expect, look and act like regular multidimensional arrays. However, as you will see, there are a couple of subtle differences. To declare a multidimensional array variable, specify each additional index using another set of square brackets. For example, the following declares a two-dimensional array variable called twoD.

```
int twoD[][] = new int[4][5];
```

- **Jagged Array:** Jagged array is a multidimensional array where member arrays are of different size. For example, we can create a 2D array where first array is of 3 elements, and is of 4 elements. Following is the example demonstrating the concept of jagged array.

```
public class Tester {
   public static void main(String[] args){
      int[][] twoDimenArray = new int[2][];

      //first row has 3 columns
      twoDimenArray[0] = new int[3];

      //second row has 4 columns
      twoDimenArray[1] = new int[4];

      int counter = 0;
      //initializing array
      for(int row=0; row < twoDimenArray.length; row++){
         for(int col=0; col < twoDimenArray[row].length; col++){
            twoDimenArray[row][col] = counter++;
         }
      }

      //printing array
      for(int row=0; row < twoDimenArray.length; row++){
         System.out.println();
         for(int col=0; col < twoDimenArray[row].length; col++){
            System.out.print(twoDimenArray[row][col] + " ");
         }
      }
   }
}
```

Output:

```
0 1 2
3 4 5 6
```

Command Line Argument

The java command-line argument is an argument i.e. passed at the time of running the java program. The arguments passed from the console can be received in the java program and it can be used as an input. So, it provides a convenient way to check the behavior of the program for the different values. You can pass N (1,2,3 and so on) numbers of arguments from the command prompt.

OOPs Concept

- **Class:** Collection of objects is called class. It is a logical entity. A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.
- **Object:** Any entity that has state and behavior is known as an object. For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical. An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory. Objects can communicate without knowing the details of each other's data or code. The only necessary thing is the type of message accepted and the type of response returned by the objects.

- **Encapsulation:** Binding (or wrapping) code and data together into a single unit are known as encapsulation. For example, a capsule, it is wrapped with different medicines. A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.
- **Inheritance:** When one object acquires all the properties and behaviors of a parent object, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.
- **Polymorphism:** If one task is performed in different ways, it is known as polymorphism. For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc. In Java, we use method overloading and method overriding to achieve polymorphism. Another example can be to speak something; for example, a cat speaks meow, dog barks woof, etc.
- **Constructor:** In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory. It is a special type of method which is used to initialize the object. Every time an object is created using the new() keyword, at least one constructor is called. It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default. There are two types of constructors in Java: no-arg constructor, and parameterized constructor. There are two types of constructors in Java Default constructor (no-arg constructor), and Parameterized constructor.

Static and Non-Static Members

Static Member: In Java, static members are those which belongs to the class and you can access these members without instantiating the class. The static keyword can be used with methods, fields, classes (inner/nested), blocks.

- **Static Methods:** You can create a static method by using the keyword static. Static methods can access only static fields, methods. To access static methods there is no need to instantiate the class, you can do it just using the class name as.
- **Static Variable:** If you declare any variable as static, it is known as a static variable. The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc. The static variable gets memory only once in the class area at the time of class loading.

Non-Static Members: Any variable of a class which is not static is called non-static variable/method or an instance variable.

Difference Between static and Non-Static Members

Key	Static	Non-Static
Access	A static variable can be accessed by static members as well as non-static member functions.	A non-static variable can not be accessed by static member functions.
Sharing	A static variable acts as a global variable and is shared among all the objects of the class.	A non-static variables are specific to instance object in which they are created.
Memory allocation	Static variables occupies less space and memory allocation happens once.	A non-static variable may occupy more space. Memory allocation may happen at run time.
Keyword	A static variable is declared using static keyword.	A normal variable is not required to have any special keyword.

Overloading

If a class has multiple methods having same name but different in parameters, it is known as Method Overloading. If we have to perform only one operation, having same name of the methods increases the readability of the program. Suppose you have to perform addition of the given numbers but there can be any

number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs. So, we perform method overloading to figure out the program quickly. Advantage of method overloading is, it increases the readability of the program. There are two ways to overload the method in java by changing number of arguments, and by changing the data type.

Varargs

The varrags allows the method to accept zero or multiple arguments. Before varargs either we use overloaded method or take an array as the method parameter but it was not considered good because it leads to the maintenance problem. If we don't know how many arguments we will have to pass in the method, varargs is the better approach. Advantage of Varargs is that we don't have to provide overloaded methods so less code.

Syntax of varargs:

```
public static void fun(int ... a)
{
   // method body
}
```

This syntax tells the compiler that fun( ) can be called with zero or more arguments. As a result, here, a is implicitly declared as an array of type int[]. Below is a code snippet for illustrating the above concept :

```
// Java program to demonstrate varargs
class Test1 {
      static void fun(int... a)
      {
            System.out.println("Number of arguments: "+ a.length);
            for (int i : a)
                  System.out.print(i + " ");
                        System.out.println();
                     }
      public static void main(String args[])

            // one parameter
            fun(100);
            // four parameters
            fun(1, 2, 3, 4);
            // no parameter
            fun();
      }
}
```

Output:

```
Number of arguments: 1
100
Number of arguments: 4
1 2 3 4
Number of arguments: 0
```

Unit – 2

Inheritance, Java Packages

Universal Class (Object Class)

A class can be defined as a template/blueprint that describes the behavior/state that the object of its type support. A class is a blueprint from which individual objects are created. Object class is present in java.lang package. Every class in Java is directly or indirectly derived from the Object class. If a Class does not extend any other class then it is direct child class of Object and if extends other class then it is an indirectly derived. Therefore, the Object class methods are available to all Java classes. Hence Object class acts as a root of inheritance hierarchy in any Java Program. Object Class Some Methods for Example…

- public final Class getClass()
- public int hashCode()
- public boolean equals(Object obj)
- public String toString()
- public final void notify()

Access Specifiers

- **Default Access Modifier** - No Keyword: Default access modifier means we do not explicitly declare an access modifier for a class, field, method, etc. A variable or method declared without any access control modifier is available to any other class in the same package. The fields in an interface are implicitly public static final and the methods in an interface are by default public.
- **Private Access Modifier – Private:** Methods, variables, and constructors that are declared private can only be accessed within the declared class itself. Private access modifier is the most restrictive access level. Class and interfaces cannot be private. Variables that are declared private can be accessed outside the class, if public getter methods are present in the class. Using the private modifier is the main way that an object encapsulates itself and hides data from the outside world.
- **Public Access Modifier – Public:** A class, method, constructor, interface, etc. declared public can be accessed from any other class. Therefore, fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe. However, if the public class we are trying to access is in a different package, then the public class still needs to be imported. Because of class inheritance, all public methods and variables of a class are inherited by its subclasses.
- **Protected Access Modifier – Protected:** Variables, methods, and constructors, which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class. The protected access modifier cannot be applied to class and interfaces. Methods, fields can be declared protected, however methods and fields in a interface cannot be declared protected. Protected access gives the subclass a chance to use the helper method or variable, while preventing a nonrelated class from trying to use it.

Inheritance

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system). 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 in your current class also. Inheritance represents the IS-A relationship which is also known as a parent-child relationship. Why use inheritance in java?

1)For Method Overriding (so runtime polymorphism can be achieved).
2)For Code Reusability.

Terms used in Inheritance:

- **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:** Subclass is a class which 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 which 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.

The syntax of Java Inheritance

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

Types of Inheritance:

- **Single Inheritance:** When a class inherits another class, it is known as a single inheritance. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.
- **Multilevel Inheritance:** When there is a chain of inheritance, it is known as multilevel inheritance. 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.
- **Hierarchical Inheritance:** When two or more classes inherits a single class, it is known as hierarchical inheritance. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

Note: Java does not support Multiple Inheritance.

Method Overriding: If subclass (child class) has the same method as declared in the parent class, it is known as method overriding in Java. In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding. The following are usage of Java Method Overriding:

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding:

- The method must have the same name as in the parent class
- The method must have the same parameter as in the parent class.
- There must be an IS-A relationship (inheritance).

Interface

An interface in Java is a blueprint of a class. It has static constants and abstract methods. The interface in Java is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java. In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

Java Interface also represents the IS-A relationship. It cannot be instantiated just like the abstract class. Since Java 8, we can have default and static methods in an interface. Since Java 9, we can have private methods in an interface. Why use Java interface? There are mainly three reasons to use interface. They are given below:

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

Syntax:

```
interface <interface_name> {
    // declare constant fields
    // declare methods that abstract by default.
}
```

Nested and Inner Class

Java Inner Classes: Java inner class or nested class is a class which is declared inside the class or interface. We use inner classes to logically group classes and interfaces in one place so that it can be more readable and maintainable. Additionally, it can access all the members of outer class including private data members and methods.

Advantage of java inner classes

- Nested classes represent a special type of relationship that is it can access all the members (data members and methods) of outer class including private.
- Nested classes are used to develop more readable and maintainable code because it logically group classes and interfaces in one place only.
- Code Optimization: It requires less code to write.

Syntax of Inner class:

```
class Java_Outer_class{
      //code
      class Java_Inner_class{
           //code
      }
}
```

Nested Class: There are two types of nested classes non-static and static nested classes. The non-static nested classes are also known as inner classes. Non-static nested class (inner class):

1)Member inner class
2)Anonymous inner class
3)Local inner class

Static nested class

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

Math Class

Java Math class provides several methods to work on math calculations like min(), max(), avg(), sin(), cos(), tan(), round(), ceil(), floor(), abs() etc. Unlike some of the StrictMath class numeric methods, all implementations of the equivalent function of Math class can't define to return the bit-for-bit same results. This relaxation permits implementation with better-performance where strict reproducibility is not required. If the size is int or long and the results overflow the range of value, the methods addExact(), subtractExact(), multiplyExact(), and toIntExact() throw an ArithmeticException. For other arithmetic operations like increment, decrement, divide, absolute value, and negation overflow occur only with a specific minimum or maximum value. It should be checked against the maximum and minimum value as appropriate.

Wrapper classes

The wrapper class in Java provides the mechanism to convert primitive into object and object into primitive. Since J2SE 5.0, autoboxing and unboxing feature convert primitives into objects and objects into primitives automatically. The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing.

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 in 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 in 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 package are known as wrapper classes in Java. The list of eight wrapper classes are given below:

Primitive Type	Wrapper class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Unit – 3

Exception Handling, Threading and Streams (Input and Output)

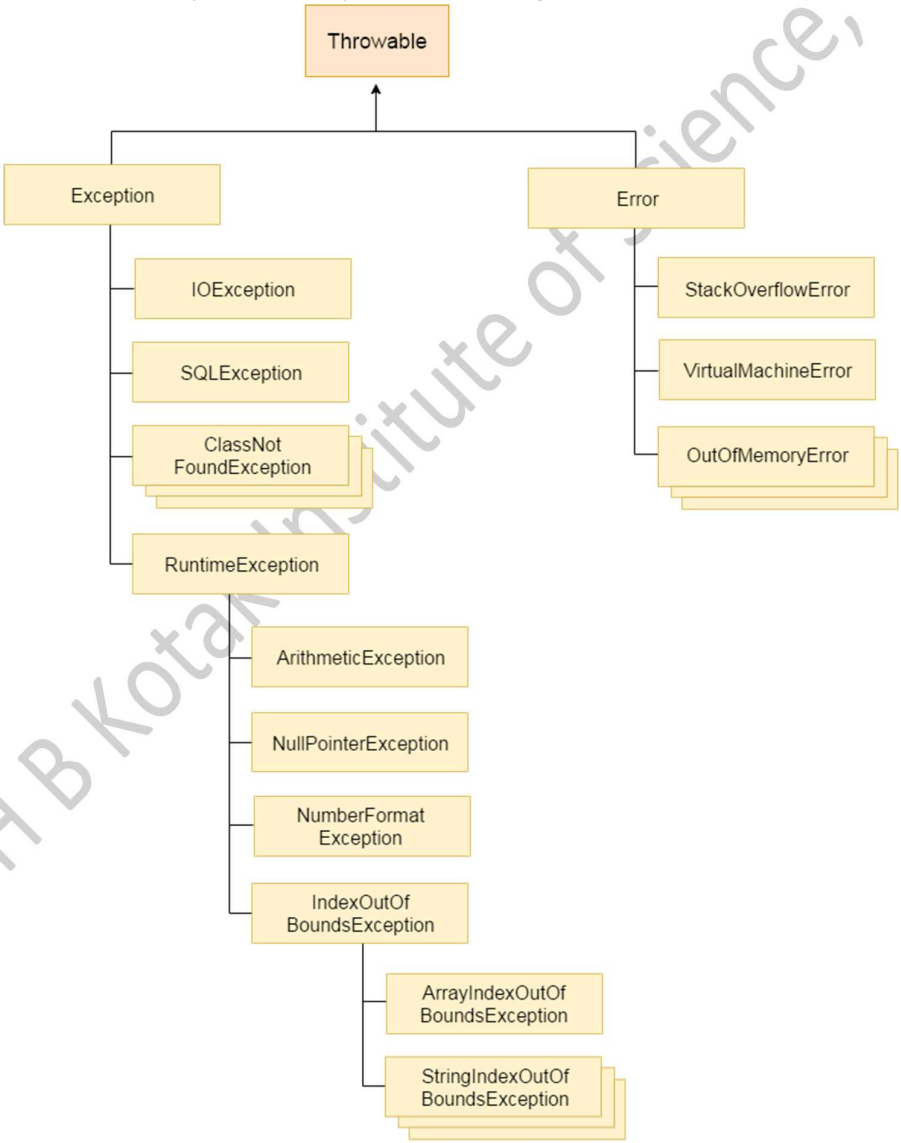
Introduction to Exception Handling

The Exception Handling in Java is one of the powerful mechanisms to handle the runtime errors so that normal flow of the application can be maintained. In this page, we will learn about Java exceptions, its type and the difference between checked and unchecked exceptions. What is Exception Handling? Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

What is Exception in Java? Dictionary Meaning: Exception is an abnormal condition. In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime. The 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 use exception handling.

Hierarchy of Java Exception classes

The java.lang.Throwable class is the root class of Java Exception hierarchy which is inherited by two subclasses: Exception and Error. A hierarchy of Java Exception classes are given below:



Types of Java Exceptions

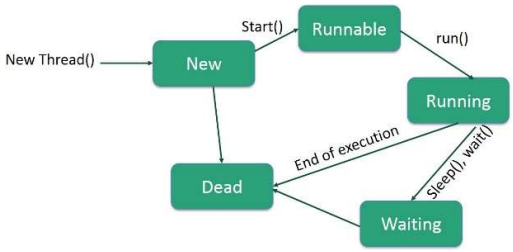
There are mainly two types of exceptions: checked and unchecked. Here, an error is considered as the unchecked exception. According to Oracle, there are three types of exceptions:

- **Checked Exception:** The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.
- **Unchecked Exception:** The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.
- **Error:** Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Try, Catch, Finally, Throw, Throws: There are 5 keywords which are used in handling exceptions in Java.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

Thread Life Cycle: 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** − A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.
- **Runnable** − After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
- **Waiting** − Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- **Timed Waiting** − A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
- **Terminated (Dead)** − A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Thread Class and its methods

The second way to create a thread is to create a new class that extends Thread class using the following two simple steps. This approach provides more flexibility in handling multiple threads created using available methods in Thread class.

- You will need to override run( ) method available in Thread class. This method provides an entry point for the thread and you will put your complete business logic inside this method Syntax:

```
public void run( )
```

- Once Thread object is created, you can start it by calling start() method, which executes a call to run() method. Syantax

```
void start( );
```

Thread Methods

Sr.	Method & Description
1	**public void start()** Starts the thread in a separate path of execution, then invokes the run() method on this Thread object.
2	**public void run()** If this Thread object was instantiated using a separate Runnable target, the run() method is invoked on that Runnable object.
3	**public final void setName(String name)** Changes the name of the Thread object. There is also a getName() method for retrieving the name.
4	**public final void setPriority(int priority)** Sets the priority of this Thread object. The possible values are between 1 and 10.
5	**public final void setDaemon(boolean on)** A parameter of true denotes this Thread as a daemon thread.
6	**public final void join(long millisec)** The current thread invokes this method on a second thread, causing the current thread to block until the second thread terminates or the specified number of milliseconds passes.
7	**public void interrupt()** Interrupts this thread, causing it to continue execution if it was blocked for any reason.
8	**public final boolean isAlive()** Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion.

Thread Synchronization

When we start two or more threads within a program, there may be a situation when multiple threads try to access the same resource and finally they can produce unforeseen result due to concurrency issues. For example, if multiple threads try to write within a same file then they may corrupt the data because one of the threads can override data or while one thread is opening the same file at the same time another thread might be closing the same file. So, there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time. This is implemented using a concept called monitors. Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor. Java programming language provides a very handy way of creating threads and synchronizing their task by using synchronized blocks. You keep shared resources within this block. Following is the general form of the synchronized statement –

```
synchronized(objectidentifier) {
   // Access shared variables and other shared resources
}
```

Multithreading Example with Synchronization

```
class PrintDemo {
```

```
        public void printCount() {
           try {
              for(int i = 5; i > 0; i--) {
                 System.out.println("Counter   ---   "  + i );
              }
           } catch (Exception e) {
              System.out.println("Thread  interrupted.");
           }
        }
     }
     class ThreadDemo extends Thread {
        private Thread t;
        private String threadName;
        PrintDemo  PD;
        ThreadDemo( String name,  PrintDemo pd) {
           threadName = name;
           PD = pd;
        }
        public void run() {
           synchronized(PD) {
              PD.printCount();
           }
           System.out.println("Thread " +  threadName + " exiting.");
        }
        public void start () {
           System.out.println("Starting " +  threadName );
           if (t == null) {
              t = new Thread (this, threadName);
              t.start ();
           }
        }
     }
     public class TestThread {
        public static void main(String args[]) {
           PrintDemo PD = new PrintDemo();
           ThreadDemo T1 = new ThreadDemo( "Thread - 1 ", PD );
           ThreadDemo T2 = new ThreadDemo( "Thread - 2 ", PD );
           T1.start();
           T2.start();
           // wait for threads to end
           try {
              T1.join();
              T2.join();
           } catch ( Exception e) {
              System.out.println("Interrupted");
           }
        }
     }
```

Deamon Thread, Non-Deamon Thread

Deamon Thread: A Daemon thread is a background service thread which runs as a low priority thread and performs background operations like garbage collection. JVM exits if only daemon threads are remaining. The setDaemon() method of the Thread class is used to mark/set a particular thread as either a daemon thread or a user thread. The Java Virtual Machine exits when the only threads running are all daemon threads. This method must be called before the thread is started. Example:

```
class adminThread extends Thread {
   adminThread() {
      setDaemon(false);
   }
   public void run() {
      boolean d = isDaemon();
      System.out.println("daemon = " + d);
   }
}
public class Tester {
   public static void main(String[] args) throws Exception {
      Thread thread = new adminThread();
      System.out.println("thread = " + thread.currentThread());
      thread.setDaemon(false);
      thread.start();
   }
}
```

Output

```
thread = Thread[main,5,main]
daemon = false
```

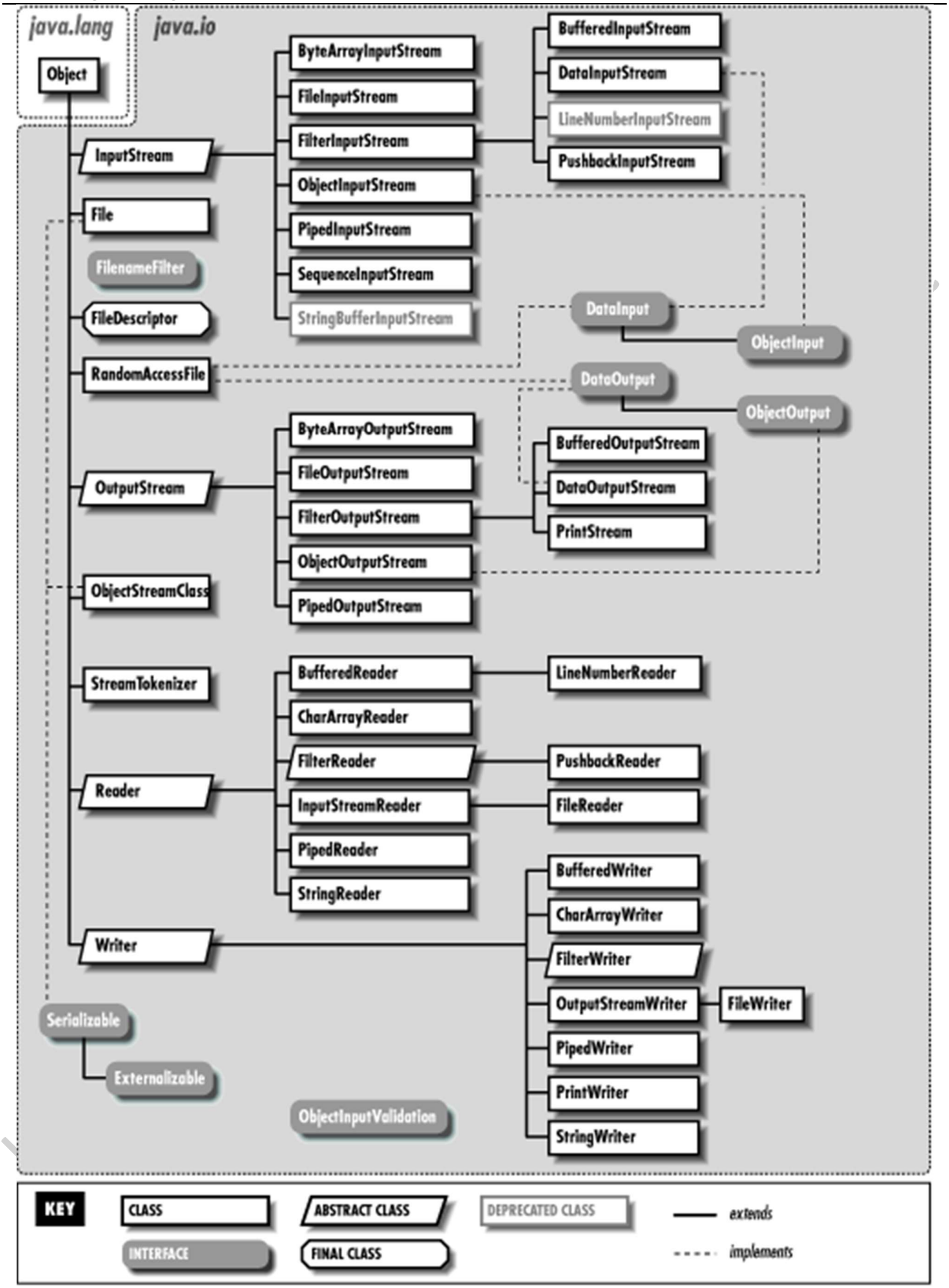
Non - Deamon Thread: The daemon threads are typically used to perform services for user threads. The main() method of the application thread is a user thread (non-daemon thread). The JVM doesn't terminate unless all the user thread (non-daemon) terminates. We can explicitly specify a thread created by a user thread to be a daemon thread by calling setDaemon(true). To determine if a thread is a daemon thread by using the method isDaemon().

Stream

A stream can be defined as a sequence of data. There are two kinds of Streams:

- **InPutStream** − The InputStream is used to read data from a source.
- **OutPutStream** − The OutputStream is used for writing data to a destination.

Java provides strong but flexible support for I/O related to files and networks with **java.io** package. The java.io package containing classes and interfaces related with I/O process. Basically there are two category of classes and interfaces: one is Byte Stream and another is Character Stream the following diagram describe hierarchy of available classes and interface in java.io pckage:



Character Streams: Java Byte streams are used to perform input and output of 8-bit bytes, whereas Java Character streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are, FileReader and FileWriter. Though

internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

Byte Streams: Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are, FileInputStream and FileOutputStream.

RandomAccessFile Class: The Java.io.RandomAccessFile class file behaves like a large array of bytes stored in the file system.Instances of this class support both reading and writing to a random access file. Following is the declaration for Java.io.RandomAccessFile class −

```
public class RandomAccessFile extends Object
implements DataOutput, DataInput, Closeable
```

Stream Tokenizer Class: The Java.io.StreamTokenizer class takes an input stream and parses it into "tokens", allowing the tokens to be read one at a time. The stream tokenizer can recognize identifiers, numbers, quoted strings, and various comment styles. Following is the declaration for Java.io.StreamTokenizer class −

```
public class StreamTokenizer extends Object
```

Unit 4

JavaFx Basics and Event-driven programming and animations

Basic Structure of JAVAFX program

JavaFX was formerly developed by Chris Oliver. At that time, he was serving for a company named See Beyond Technology Corporation. Initially, the JavaFX project was recognized as Form Follows Functions (F3). This project was designed with the aim of providing richer interfaces for developing GUI applications. Later in June 2005, Sun Micro-systems took the F3 project and changed its name from F3 to JavaFX. Timeline of JavaFX:

- 2005 – Sun Microsystems took over the See Beyond company in June 2005 and acquired the F3 project as JavaFX.
- 2007 – JavaFX was officially declared at Java One, a worldwide web conference that is held yearly.
- 2008 – Net Beans integration with JavaFX was made open. The Java Standard Development Kit for JavaFX 1.0 was also released in the same year.
- 2009 – The next version of JavaFX was released, i.e., JavaFX 1.2, and the support for JavaFX Mobile was also introduced. In the same year only, Oracle Corporation also acquired Sun Microsystems.
- 2010 – JavaFX version 1.3 was released in 2010.
- 2011 – In 2011, JavaFX version 2.0 came out.
- 2012 – The support for JavaFX Mac OS for desktop was introduced.
- 2014 – The most advanced version of JavaFX, i.e., JavaFX 8, was released as an indispensable part of Java on the 18th of March 2014.

Need of JavaFX: Before JavaFX, developing the client base applications was a very complicated and cumbersome task. Programmers and developers used to require several libraries for adding various functionalities like media, UI controls, animations and effects, 2D and 3D shapes, etc., in their applications. This issue was resolved when JavaFX came into the picture, which changed the whole scenario of the development of web applications by bringing in all the peculiarities into one single library. Apart from this, the programmers can also utilize all the existing perks of the older libraries like Java Swing and Advanced Windowing Tool Kit. JavaFX also displays a valuable collection of graphics and various media APIs, which can further help in designing smooth applications. JavaFX also leverages the improved Graphical Processing Unit (GUI) with the help of hardware-accelerated graphics. If a developer wants to combine the graphics animations and UI control in their applications, then they can use the various interfaces provided by the JavaFX.

What is JavaFX?

JavaFX is a Java library used to build **Rich Internet Applications (RIA)**. The applications written using this library can run consistently across multiple platforms. The applications developed using JavaFX can run on various devices such as Desktop Computers, Mobile Phones, TVs, Tablets, etc. To develop GUI Applications using Java programming language, the programmers rely on libraries such as Advanced Windowing Toolkit and Swing. After the advent of JavaFX, these Java programmers can now develop GUI applications effectively with rich content.

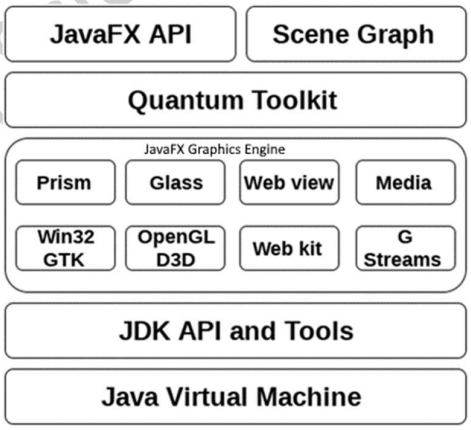
Features of JavaFX: Following are some of the important features of JavaFX

- Written in Java − The JavaFX library is written in Java and is available for the languages that can be executed on a JVM, which include − Java, Groovy and JRuby. These JavaFX applications are also platform independent.
- FXML − JavaFX features a language known as FXML, which is a HTML like declarative markup language. The sole purpose of this language is to define a user Interface.
- Scene Builder − JavaFX provides an application named Scene Builder. On integrating this application in IDE's such as Eclipse and NetBeans, the users can access a drag and drop design interface, which is used to develop FXML applications (just like Swing Drag & Drop and DreamWeaver Applications).
- Swing Interoperability − In a JavaFX application, you can embed Swing content using the Swing Node class. Similarly, you can update the existing Swing applications with JavaFX features like embedded web content and rich graphics media.

- Built-in UI controls − JavaFX library caters UI controls using which we can develop a full-featured application.
- CSS like Styling − JavaFX provides a CSS like styling. By using this, you can improve the design of your application with a simple knowledge of CSS.
- Canvas and Printing API − JavaFX provides Canvas, an immediate mode style of rendering API. Within the package javafx.scene.canvas it holds a set of classes for canvas, using which we can draw directly within an area of the JavaFX scene. JavaFX also provides classes for Printing purposes in the package javafx.print.
- Rich set of API's − JavaFX library provides a rich set of API's to develop GUI applications, 2D and 3D graphics, etc. This set of API's also includes capabilities of Java platform. Therefore, using this API, you can access the features of Java languages such as Generics, Annotations, Multithreading, and Lambda Expressions. The traditional Java Collections library was enhanced and concepts like observable lists and maps were included in it. Using these, the users can observe the changes in the data models.
- Integrated Graphics library − JavaFX provides classes for 2d and 3d graphics.
- Graphics pipeline − JavaFX supports graphics based on the Hardware-accelerated graphics pipeline known as Prism. When used with a supported Graphic Card or GPU it offers smooth graphics. In case the system does not support graphic card then prism defaults to the software rendering stack.

JavaFX Architecture

The following image shows the complete architecture of JavaFX platform. There are various built-in components which are interconnected with each other. However, JavaFX contains a rich set of APIs which are more than enough to develop rich internet applications which run consistently across many platforms. JavaFX has numerous built-in elements that are interconnected with each other. JavaFX library comprises a valuable collection of APIs, classes, and interfaces that are more than sufficient to produce rich internet applications and GUI applications with intense graphics that can run consistently over multiple platforms. As we can see in the figure that, JavaFX architecture comprises many different components. These components are briefly described as follows:

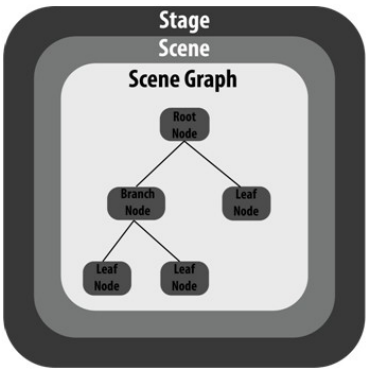


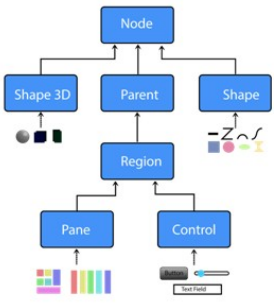
- **JavaFX API** – The topmost layer of JavaFX architecture holds a JavaFX public API that implements all the required classes that are capable of producing a full-featured JavaFX application with rich graphics. The list of all the important packages of this API is as follows.
  - javafx.animation: It includes classes that are used to combine transition-based animations such as fill, fade, rotate, scale and translation, to the JavaFX nodes (collection of nodes makes a scene graph).
  - javafx.css − It comprises classes that are used to append CSS–like styling to the JavaFX GUI applications.
  - javafx.geometry − It contains classes that are used to represent 2D figures and execute methods on them.
  - javafx.scene − This package of JavaFX API implements classes and interfaces to establish the scene graph. In extension, it also renders sub-packages such as canvas, chart, control, effect, image, input, layout, media, paint, shape, text, transform, web, etc. These are the diverse elements that sustain this precious API of JavaFX.
  - javafx.application − This package includes a collection of classes that are responsible for the life cycle of the JavaFX application.
  - javafx.event − It includes classes and interfaces that are used to perform and manage JavaFX events.

- javafx.stage − This package of JavaFX API accommodates the top-level container classes used for the JavaFX application.

- **Scene Graph** – A Scene Graph is the starting point of the development of any of the GUI Applications. In JavaFX, all the GUI Applications are made using a Scene Graph only. The Scene Graph includes the primitives of the rich internet applications that are known as nodes. In simple words, a single component in a scene graph is known as a node. In general, a scene graph is made up of a collection of nodes. All these nodes are organized in the form of a hierarchical tree that describes all of the visual components of the application's user interface (UI). A node instance can be appended to a scene graph only once. The nodes of a scene graph can have numerous segments like Effects, Opacity, Transforms, Event Handlers, Application Specific States. The nodes are of three general types.
- **Graphics Engine** - The JavaFX graphics engine provides the graphics support to the the scene graph. It basically supports 2D as well as 3D graphics both. It provides the software rendering when the graphics hardware present on the system is not able to support hardware accelerated rendering. The two graphics accelerated pipelines in the JavaFX are:
  - **Prism:** prism can be seen as High Performance hardware-accelerated graphics pipeline. It has the capability to render both 2D and 3D graphics. Prism implements different ways to render graphics on different platforms. DirectX 9 on windows XP or vista, DirectX 11 on windows 7, OpenGL on Mac, Linux and embedded, Java 2D when hardware acceleration is not possible
  - **Quantum Tool kit:** Quantum Tool Kit is used to bind prism and glass windowing tool kit together and makes them available for the above layers in stack.
  - **Glass Windowing tool kit:** It is present on the lowest level of JavaFX graphics stack. It basically can be seen as a platform dependent layer which works as an interface between JavaFX platform and native operating system. It is responsible for providing the operating system services such as managing the windows, timers, event queues and surfaces.
  - **Web View:** We can also embed the HTML content to a JavaFX scene graph. For this purpose, JavaFX uses a component called web view. Web view uses web kit which is an internal open source browser and can render HTM5, DOM, CSS, SVG and JavaScript. Using web view, we can render the HTML content from JavaFX application , and also apply some CSS styles to the user interface.
  - **Media Engine:** By using Media engine, the JavaFX application can support the playback of audio and video media files. JavaFX media engine depends upon an open source engine called as G Streamer. The package javafx.scene.media contains all the classes and interfaces that can provide media functionalities to JavaFX applications.

JavaFX Application Structure: JavaFX application is divided hierarchically into three main components known as Stage, Scene and nodes. We need to import javafx.application.Application class in every JavaFX application.





- **Stage:** Stage in a JavaFX application is similar to the Frame in a Swing Application. It acts like a container for all the JavaFX objects. Primary Stage is created internally by the platform. Other stages can further be created by the application. The object of primary stage is passed to start method. We need to call show method on the primary stage object in order to show our primary stage. However, we can add various objects to this primary stage. The objects can only be added in a hierarchical way i.e. first, scene graph will be added to this primaryStage and then that scene graph may contain the nodes. A node may be any object of the user's interface like text area, buttons, shapes, media, etc.

- **Scene:** Scene actually holds all the physical contents (nodes) of a JavaFX application. Javafx.scene.Scene class provides all the methods to deal with a scene object. Creating scene is necessary in order to visualize the contents on the stage. At one instance, the scene object can only be added to one stage. In order to implement Scene in our JavaFX application, we must import javafx.scene package in our code. The Scene can be created by creating the Scene class object and passing the layout object into the Scene class constructor. We will discuss Scene class and its method later in detail.
- **Scene Graph:** Scene Graph exists at the lowest level of the hierarchy. It can be seen as the collection of various nodes. A node is the element which is visualized on the stage. It can be any button, text box, layout, image, radio button, check box, etc.

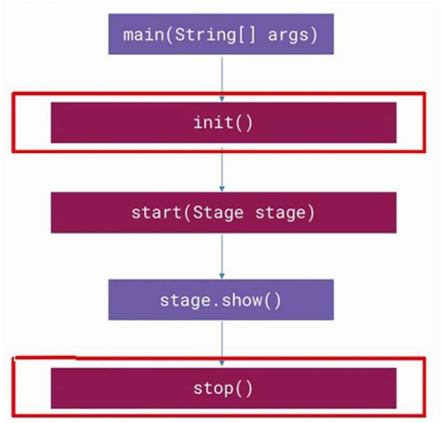
The nodes are implemented in a tree kind of structure. There is always one root in the scene graph. This will act as a parent node for all the other nodes present in the scene graph. However, this node may be any of the layouts available in the JavaFX system. The leaf nodes exist at the lowest level in the tree hierarchy. Each of the node present in the scene graphs represents classes of javafx.scene package therefore we need to import the package into our application in order to create a full featured javafx application.

LifeCycle of a JavaFX Application: javafx.application.Application class in every JavaFX application. This provides the following life cycle methods for JavaFX application.

- public void init()
- public abstract void start(Stage primaryStage)
- public void stop()

There are in total three life cycle methods of a JavaFX Application class. These methods are –

- **start() –** The start() method is the entry point method of the JavaFX application where all the graphics code of JavaFX is to be written.
- **init() –** The init() method is an empty method that can be overridden. In this method, the user cannot create a stage or a scene.
- **stop() –** The stop() method is an empty method that can also be overridden, just like the init() method. In this method, the user can write the code to halt the application.



Other than these methods, the JavaFX application also implements a static method known as **launch()**. This launch() method is used to launch the JavaFX application. As stated earlier, the launch() method is static, the user should call it from a static method only. Generally, that static method, which calls the launch() method, is the main() method only. Whenever a user launches a JavaFX application, there are few actions that are carried out in a particular manner only. The following is the order given in which a JavaFX application is launched. Firstly, an instance of the application class is created. After that, the init() method is called. After the init() method, the start() method is called. After calling the start() method, the launcher waits for the JavaFX application to end and then calls the stop() method.

Panes

Layouts are the top-level container classes that describe the UI styles for objects of the scene graph. The JavaFX layout can be seen as the father node to all the separate nodes. JavaFX presents several layout panes that promote various styles of layouts. In JavaFX, Layout describes the process in which the elements are to be viewed on the screen. It primarily establishes the scene-graph nodes. There are various built-in layout panes in JavaFX that are HBox, VBox, StackPane, FlowBox, AnchorPane, etc. Each Built-in layout is represented by a separate class that requires to be instantiated in order to implement that specific layout pane. All these classes

belong to the javafx.scene.layout package. The javafx.scene.layout.Pane class is the root class for all the built-in classes of JavaFX layouts.

Shapes

In general, a two-dimensional shape can be defined as the geometrical figure that can be drawn on the coordinate system consist of X and Y planes. However, this is different from 3D shapes in the sense that each point of the 2D shape always consists of two coordinates (X,Y). Using JavaFX, we can create 2D shapes such as Line, Rectangle, Circle, Ellipse, Polygon, Cubic Curve, quad curve, Arc, etc. The class javafx.scene.shape.Shape is the base class for all the shape classes.

In some of the applications, we need to show two dimensional shapes to the user. However, JavaFX provides the flexibility to create our own 2D shapes on the screen . There are various classes which can be used to implement 2D shapes in our application. All these classes resides in javafx.scene.shape package. The following table consists of the JavaFX shape classes along with their descriptions.

Shape	Description
Line	In general, Line is the geometrical figure which joins two (X,Y) points on 2D coordinate system. In JavaFX, **javafx.scene.shape.Line** class needs to be instantiated in order to create lines.
Rectangle	In general, Rectangle is the geometrical figure with two pairs of two equal sides and four right angles at their joint. In JavaFX, **javafx.scene.shape.Rectangle** class needs to be instantiated in order to create Rectangles.
Ellipse	In general, ellipse can be defined as a curve with two focal points. The sum of the distances to the focal points are constant from each point of the ellipse. In JavaFX. **javafx.scene.shape.Ellipse** class needs to be instantiated in order to create Ellipse.
Arc	Arc can be defined as the part of the circumference of the circle of ellipse. In JavaFX, **javafx.scene.shape.Arc** class needs to be instantiated in order to create Arcs.
Circle	A circle is the special type of Ellipse having both the focal points at the same location. In JavaFX, Circle can be created by instantiating **javafx.scene.shape.Circle** class.
Polygon	Polygon is a geometrical figure that can be created by joining the multiple Co-planner line segments. In JavaFX, **javafx.scene.shape**. Pollygon class needs to be instantiated in order to create polygon.
Cubic Curve	A Cubic curve is a curve of degree 3 in the XY plane. In Javafx, **javafx.scene.shape.CubicCurve** class needs to be instantiated in order to create Cubic Curves.
Quad Curve	A Quad Curve is a curve of degree 2 in the XY plane. In JavaFX, **javafx.scene.shape.QuadCurve** class needs to be instantiated in order to create QuadCurve.

JavaFX enables us to create the three dimensional shapes. There are the classes defined in javafx.scene.shape package which provides all the methods to deal with the 3D shapes. Such classes are Box, Cylinder and sphere. The class Shape3D of the package javafx.scene.shape is the base class of the all the 3D shape classes in javafx.

A three dimensional Shape can be defined as a solid geometrical object that is to be drawn on XYZ coordinate system. 3D shapes are different from the 2D shapes in the sense that the 3D shapes always need to have an extra coordinate value Z in order to be drawn on a coordinate system. Examples of 3D shapes are cylinders, spheres, boxes, cubes, pyramids, etc. However, JavaFX provides classes to create spheres, cylinders and boxes.

Property binding

In some of the cases, we need to provide the text based information on the interface of our application. JavaFX library provides a class named javafx.scene.text.Text for this purpose. This class provides various methods to alter various properties of the text. We just need to instantiate this class to implement text in our application. The properties of JavaFX Text are described in the table below.

Property	Description	Setter Methods
boundstype	This property is of object type. It determines the way in which the bounds of the text is being calculated.	setBoundsType(TextBoundsType value)
font	Font of the text.	setFont(Font value)
fontsmoothingType	Defines the requested smoothing type for the font.	setFontSmoothingType(FontSmoothingType value)
linespacing	Vertical space in pixels between the lines. It is double type property.	setLineSpacing(double spacing)
strikethrough	This is a boolean type property. We can put a line through the text by setting this property to true.	setStrikeThrough(boolean value)
textalignment	Horizontal Text alignment	setTextAlignment(TextAlignment value)
textorigin	Origin of text coordinate system in local coordinate system.	setTextOrigin(VPos value)
text	It is a string type property. It defines the text string which is to be displayed.	setText(String value)
underline	It is a boolean type property. We can underline the text by setting this property to true.	setUnderLine(boolean value)
wrappingwidth	Width limit for the text from where the text is to	setWrappingWidth(double value)

	be wrapped. It is a double type property.	
x	X coordinate of the text	setX(double value)
y	Y coordinate of the text	setY(double value)

The class javafx.scene.text.Text needs to be instantiated in order to create the text node. Use the setter method setText(string) to set the string as a text for the text class object. Follow the syntax given below to instantiate the Text class.

```
Text <text_Object> = new Text();
text.setText(<string-text>);
```

Font class

JavaFX enables us to apply various fonts to the text nodes. We just need to set the property font of the Text class by using the setter method setFont(). This method accepts the object of Font class. The class Font belongs the package javafx.scene.text. It contains a static method named font(). This returns an object of Font type which will be passed as an argument into the setFont() method of Text class. The method Font.font() accepts the following parameters.

- Family: it represents the family of the font. It is of string type and should be an appropriate font family present in the system.
- Weight: this Font class property is for the weight of the font. There are 9 values which can be used as the font weight. The values are FontWeight.BLACK, BOLD, EXTRA_BOLD, EXTRA_LIGHT, LIGHT, MEDIUM, NORMAL, SEMI_BOLD, THIN.
- Posture: this Font class property represents the posture of the font. It can be either FontPosture.ITALIC or FontPosture.REGULAR.
- Size: this is a double type property. It is used to set the size of the font.

The Syntax of the method setFont() is given below.

```
<text_object>.setFont(Font.font(<String font_family>, <FontWeight>, <FontPosture>, <FontSize>)
```

Color class

Stroke means the padding at the boundary of the text. JavaFX allows us to apply stroke and colors to the text. javafx.scene.text.Text class provides a method named setStroke() which accepts the Paint class object as an argument. Just pass the color which will be painted on the stroke. We can also set the width of the stroke by passing a width value of double type into setStrokeWidth() method. To set the color of the Text, javafx.scene.text.Text class provides another method named setFill(). We just need to pass the color which is to be filled in the text. The following example illustrates the functions of above mentioned methods.

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.scene.text.FontPosture;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;
import javafx.stage.Stage;
public class TextExample extends Application{
    @Override
    publicvoid start(Stage primaryStage) throws Exception {
        // TODO Auto-generated method stub
        Text text = new Text();
        text.setX(100);
        text.setY(20);
```

```
            text.setFont(Font.font("AbyssinicaSIL",FontWeight.BOLD,
                    FontPosture.REGULAR,30));
            text.setFill(Color.BLUE);// setting colour of the text to blue
            text.setStroke(Color.BLACK); // setting the stroke for the text
            text.setStrokeWidth(1); // setting stroke width to 2
            text.setText("Welcome to JavaFX");
            Group root = new Group();
            Scene scene = new Scene(root,500,200);
            root.getChildren().add(text);
            primaryStage.setScene(scene);
            primaryStage.setTitle("Text Example");
            primaryStage.show();
        }
        public static void main(String[] args) {
            launch(args);
        }
    }
```

Image and Image-View class

Images are one of the most common elements that are used on any application, including JavaFX applications. An image can be in various forms like photographs, graphics or individual video frames etc. There are various digital formats of images which are also supported by JavaFX, they are: BMP, GIF, JPEG, and PNG. You can load and modify images of all the formats mentioned above using the classes provided by JavaFX in the package javafx.scene.image. You can load an image in JavaFX by instantiating the class named Image of the package javafx.scene.image. To the constructor of the Image class, you have to pass either of the following as the image source −

- An InputStream object of the image to be loaded or,
- A string variable holding the URL for the image.

An image can also be resized while they are being loaded, in order to reduce its memory storage amount. This can be done by passing following optional parameters to the constructor of Image class.

- requestedWidth to set the target width of the displayed image.
- requestedHeight to set the target height of the displayed image.
- preserveRatio is a boolean value that specifies whether the aspect ratio of the final image must remain the same as the original image or not.
- smooth represents the quality of filtering applied on the image.
- backgroundLoading represents whether the image needs to be loaded in the background or not.

Once the image is loaded, you can view it by instantiating the **ImageView class**. Same image instance can be used by multiple ImageView classes to display it. Following is the syntax to load and view an Image:

```
//Passing FileInputStream object as a parameter
FileInputStream inputstream = new FileInputStream("C:\\images\\image.jpg");
Image image = new Image(inputstream);

//Loading image from URL
//Image image = new Image(new FileInputStream("url for the image));
```

After loading the image, you can set the view for the image by instantiating the ImageView class and passing the image to its constructor as follows −

```
ImageView imageView = new ImageView(image);
```

Following is an example which demonstrates how to load an image in JavaFX and set the view. Save this code in a file with the name ImageExample.java.

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import javafx.application.Application;
```

```
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.stage.Stage;
public class ImageExample extends Application {
   @Override
   public void start(Stage stage) throws FileNotFoundException {
      //Creating an image
      Image image = new Image(new FileInputStream("path of the
image"));
      //Setting the image view
      ImageView imageView = new ImageView(image);
      //Setting the position of the image
      imageView.setX(50);
      imageView.setY(25);
      //setting the fit height and width of the image view
      imageView.setFitHeight(455);
      imageView.setFitWidth(500);
      //Setting the preserve ratio of the image view
      imageView.setPreserveRatio(true);
      //Creating a Group object
      Group root = new Group(imageView);
      //Creating a scene object
      Scene scene = new Scene(root, 600, 500);
      //Setting title to the Stage
      stage.setTitle("Loading an image");
      //Adding scene to the stage
      stage.setScene(scene);
      //Displaying the contents of the stage
      stage.show();
   }
   public static void main(String args[]) {
      launch(args);
   }
}
```

JavaFX Layouts

Layouts are the top level container classes that define the UI styles for scene graph objects. Layout can be seen as the parent node to all the other nodes. JavaFX provides various layout panes that support different styles of layouts. In JavaFX, Layout defines the way in which the components are to be seen on the stage. It basically organizes the scene-graph nodes. We have several built-in layout panes in JavaFX that are HBox, VBox, StackPane, FlowBox, AnchorPane, etc. Each Built-in layout is represented by a separate class which needs to be instantiated in order to implement that particular layout pane. All these classes belong to javafx.scene.layout package. javafx.scene.layout.Pane class is the base class for all the built-in layout classes in JavaFX. javafx.scene.layout Package provides various classes that represents the layouts. The classes are described in the table below.

Class	Description
BorderPane	Organizes nodes in top, left, right, centre and the bottom of the screen.
FlowPane	Organizes the nodes in the horizontal rows according to the available horizontal spaces. Wraps the nodes to the next line if the horizontal space is less than the total width of the nodes
GridPane	Organizes the nodes in the form of rows and columns.
HBox	Organizes the nodes in a single row.
Pane	It is the base class for all the layout classes.
StackPane	Organizes nodes in the form of a stack i.e. one onto another
VBox	Organizes nodes in a vertical column.

Steps to create layout: In order to create the layouts, we need to follow the following steps.

1. Instantiate the respective layout class, for example, HBox root = new HBox();
2. Setting the properties for the layout, for example, root.setSpacing(20);
3. Adding nodes to the layout object, for example, root.getChildren().addAll(<NodeObjects>);

Events and Events sources

In JavaFX, we can develop GUI applications, web applications and graphical applications. In such applications, whenever a user interacts with the application (nodes), an event is said to have been occurred. For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page are the activities that causes an event to happen. The events can be broadly classified into the following two categories −

- **Foreground Events** − Those events which require the direct interaction of a user. They are generated as consequences of a person interacting with the graphical components in a Graphical User Interface. For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page, etc.
- **Background Events** − Those events that don't require the interaction of end-user are known as background events. The operating system interruptions, hardware or software failure, timer expiry, operation completion are the example of background events.

JavaFX provides support to handle a wide variety of events. The class named Event of the package javafx.event is the base class for an event. An instance of any of its subclass is an event. JavaFX provides a wide variety of events. Some of them are listed below.

- **Mouse Event** − This is an input event that occurs when a mouse is clicked. It is represented by the class named MouseEvent. It includes actions like mouse clicked, mouse pressed, mouse released, mouse moved, mouse entered target, mouse exited target, etc.
- **Key Event** − This is an input event that indicates the key stroke occurred on a node. It is represented by the class named KeyEvent. This event includes actions like key pressed, key released and key typed.
- **Drag Event** − This is an input event which occurs when the mouse is dragged. It is represented by the class named DragEvent. It includes actions like drag entered, drag dropped, drag entered target, drag exited target, drag over, etc.
- **Window Event** − This is an event related to window showing/hiding actions. It is represented by the class named WindowEvent. It includes actions like window hiding, window shown, window hidden, window showing, etc.

Registering Handlers and Handling Events

In JavaFX, events are primarily used to inform the application regarding the actions chosen by the user. JavaFX implements the tool to achieve the events, route the event to its target, and granting the application to handle the events. JavaFX presents the class javafx.event.Event which includes all the subclasses describing the varieties of events that can be created in JavaFX. Any event is an instance of the class Event or any of its subclasses. There are several events in JavaFX i.e. MouseEvent, KeyEvent, ScrollEvent, DragEvent, etc. A user can further specify their personal event by inheriting the class javafx.event.Event. Most of these methods exist in the classes like Node, Scene, Window, etc., and they are available to all their sub classes. Following table describes various convenience methods that can be used on different events:

User Action	Event Type	EventHandler Properties
Pressing, releasing, or typing a key on keyboard.	KeyEvent	onKeypressed onKeyReleased onKeyTyped
Moving, Clicking, or dragging the mouse.	MouseEvent	onMouseClicked onMouseMoved onMousePressed onMouseReleased onMouseEntered onMouseExited
Pressing, Dragging, and Releasing of the mouse button.	MouseDragEvent	onMouseDragged onMouseDragEntered onMouseDragExited onMouseDragged onMouseDragOver onMouseDragReleased
Generating, Changing, Removing or Committing input from an alternate method.	InputMethodEvent	onInputMethodTextChanged
Performing Drag and Drop actions supported by the platform.	DragEvent	onDragDetected onDragDone onDragDropped onDragEntered onDragExited onDragOver
Scrolling an object.	ScrollEvent	onScroll onScrollStarted onScrollFinished
Rotating an object.	RotateEvent	onRotate onRotationFinished onRotationStarted
Swiping an object upwards, downwards, to the right and left.	SwipeEvent	onSwipeUP onSwipeDown onSwipeLeft onSwipeRight
Touching an object.	TouchEvent	onTouchMoved onTouchReleased

User Action	Event Type	EventHandler Properties
		onTouchStationary
Zooming on an object.	ZoomEvent	onZoom onZoomStarted onZoomFinished
Requesting Context Menu.	ContextMenuEvent	onContextMenuRequested
Pressing a button, showing or hiding a combo box, selecting a menu item.	ActionEvent	
Editing an item in a list.	ListView.EditEvent	
Editing an item in a table.	TableColumn.CellEditEvent	
Editing an item in a tree.	TreeView.EditEvent	
Encountering an error in a media player.	MediaErrorEvent	
Showing or Hiding Menu.	Event	
Hiding a pop-up window.	Event	
Selecting or Closing a Tab.	Event	
Showing, Minimizing, or Closing a Window.	WindowEvent	

Following is the format of Convenience methods used for registering event handlers:

```
setOnEvent-type(EventHandler<? super event-class> value)
```

For example, to add a mouse event listener to a button, you can use the convenience method setOnMouseClicked() as shown below:

```
playButton.setOnMouseClicked((new EventHandler<MouseEvent>() {
   public void handle(MouseEvent event) {
      System.out.println("Hello World");
      pathTransition.play();
   }
}));
```

Inner Classes, anonymous inner class handlers

Event handlers enable you to handle an event during the event bubbling phase of event processing. The bubbling phase of an event route is a phase where the event travels from target node to stage node. Like Event filters, a node can have one or more handlers, or no handlers at all to handle an event. If the node does not contain a handler, the event reaches the root node and the process is completed. Otherwise, if a node in event dispatch chain contains a handler, the handler is executed. A single handler can be used for more than one node and more than one event type. If an event handler for a child node does not consume the event, an event handler for a parent node enables the parent node to act on the event after a child node processes it and to provide common event processing for multiple child nodes. To add an event handler to a node, you need to register this handler using the method addEventHandler() of the Node class as shown below.

```
//Creating the mouse event handler
EventHandler<javafx.scene.input.MouseEvent> eventHandler =
```

```
        new EventHandler<javafx.scene.input.MouseEvent>() {

        @Override
        public void handle(javafx.scene.input.MouseEvent e) {
           System.out.println("Hello World");
           circle.setFill(Color.DARKSLATEBLUE);
        }
    };
    //Adding the event handler
    circle.addEventHandler(javafx.scene.input.MouseEvent.MOUSE_CLICKED,
    eventHandler);
```

In the same way, you can remove an event handler using the method removeEventHandler() as shown below –

```
circle.removeEventHandler(MouseEvent.MOUSE_CLICKED, eventHandler);
```

Listeners for observable objects

For the most part, listeners are used to trigger imperative code due to changes in the State of your application. Most often, they are used to perform operations that need to be done off the FXAT, or to invoke functionality which simply isn't designed to be Reactive. For instance, Animations are actions that need to be triggered, and this can be done through a listener. In most cases, however, it's better to implement a more direct trigger like an ActionEvent if it's possible. Sometimes a listener can be useful when it's too complicated to implement a Binding. Perhaps, if there is an arbitrarily large number of elements that need to be connected in a complicated way, attempting to install a Binding on each one may be prohibitive. In that case, having a single listener that then updates all of the dependent values might be less complicated and clearer. Before we look at the listeners, let's take a look at how the data values are wrapped into observables.

In JavaFX, the great-great-grandfather of Properties and Bindings is the Observable interface. The great-grandfather is the ObservableValue interface. These two interfaces define the most important aspect of anything observable - that it can be listened to. Let's have a quick look at both of these interfaces, and see what they do. Observable is a top level interface, so it doesn't inherit any methods from any other interfaces. It has just two methods:

```
addListener(InvalidationListener listener)
removeListener(InvalidationListener listener)
```

Animation

In general, the animation can be defined as the transition which creates the myth of motion for an object. It is the set of transformations applied on an object over the specified duration sequentially so that the object can be shown as it is in motion. This can be done by the rapid display of frames. In JavaFX, the package javafx.animation contains all the classes to apply the animations onto the nodes. All the classes of this package extend the class javafx.animation.Animation. JavaFX provides the classes for the transitions like RotateTransition, ScaleTransition, TranslateTransition, FadeTransition, FillTransition, StrokeTransition, etc. The package javafx.animation provides the classes for performing the following transitions.

Transition	Description
Rotate Transition	Rotate the Node along one of the axes over the specified duration.
Scale Transition	Animate the scaling of the node over the specified duration.

Translate Transition	Translate the node from one position to another over the specified duration.
Fade Transition	Animate the opacity of the node. It keeps updating the opacity of the node over a specified duration in order to reach a target opacity value
Fill Transition	Animate the node's fill color so that the fill color of the node fluctuates between the two color values over the specified duration.
Stroke Transition	Animate the node's stroke color so that the stroke color of the node fluctuates between the two color values over the specified duration.
Sequential Transition	Perform the list of transitions on a node in the sequential order.
Parallel Transition	Perform the list of transitions on a node in parallel.
Path Transition	Move the node along the specified path over the specified duration.

Steps for applying Animations:

1. Create the target node and configure its properties.
```
Rectangle rect = new Rectangle(120,100,100,100);
rect.setFill(Color.RED);
```

2. Instantiate the respective transition class
```
RotateTransition rotate = new RotateTransition();
```

3. Set the desired properties like duration, cycle-count, etc. for the transition.
```
rotate.setDuration(Duration.millis(1000));
rotate.setAxis(Rotate.Y_Axis);
rotate.setCycleCount(500);
```

4. Set the target node on which the transition will be applied. Use the following method for this purpose.
```
rotate.setNode(rect);
```

5. Finally, play the transition using the play() method.
```
rotate.play();
```

Unit 5

JavaFx UI controls and multimedia

JavaFX UI Controls

UI Controls are the graphical elements that allow users to interact with an application or a website. They include buttons, menus, sliders, text fields, checkboxes, radio buttons, and more. In this tutorial, we will explore the different types of UI Controls of JavaFX. let's start the discussion by introducing three main aspects of an user interface −

- **UI elements** − These are the core visual elements which the user eventually sees and interacts with. JavaFX provides a huge list of widely used and common elements varying from basic to complex, which we will cover in this tutorial.
- **Layouts** − They define how UI elements should be organized on the screen and provide a final look and feel to the GUI (Graphical User Interface). This part will be covered in the Layout chapter.
- **Behavior** − These are events which occur when the user interacts with UI elements. This part will be covered in the Event Handling chapter.

To create GUI components (controls), JavaFX supports several controls such as date picker, button text field, etc. These controls are represented by different classes of the package **javafx.scene.control**. We can create a control by instantiating its respective class.

Labeled and Label

A Label is a piece of text that describe or informs users about the functionality of other elements in the application. It helps in reducing confusion and provides clarity which leads to a better user experience. Always remember, it is a not an editable text control. In JavaFX, the label is represented by a class named Label which belongs to the javafx.scene.control package. To create a label in JavaFX application, we can use any of the below constructor −

- Label() − It is the default constructor that constructs an empty label.
- Label(String str) − It constructs a label with the predefined text.
- Label(String str, Node graph) − It constructs a new label with the specified text and graph.

To create a Label in JavaFX, follow the steps given below −

Step 1: Instantiate the Label class: As discussed earlier, we need to instantiate the Label class to create a label text. We can use either its default constructor or parameterized constructor. If we use the default one, the label text is added by using the setText() method.

```
// Instanting the Label class
Label label = new Label("Sample label");
```

Step 2: Set the required properties of Label: Just like a text node we can set the desired properties like font and font color to the label node in JavaFX using the setFont() method and setFill() method respectively.

```
// Setting font to the label
Font font = Font.font("Brush Script MT", FontWeight.BOLD,
FontPosture.REGULAR, 25);
label.setFont(font);
// Filling color to the label
label.setTextFill(Color.BROWN);
```

Step 3: Launching Application: Once the Label is created and its properties are set, define a group object to hold the label. Next, create a Scene object by passing the group obejct and the dimensions of the Scene to its constructor. Then, set the stage and launch the application to display the result.

Button

JavaFX button control is represented by javafx.scene.control.Button class. A button is a component that can control the behaviour of the Application. An event is generated whenever the button gets clicked. Button can be created by instantiating Button class. Use the following line to create button object.

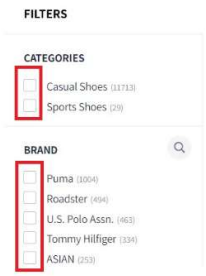
```
Button btn = new Button("My Button");
```

To visualize the button on the screen, we must attach it to the scene object. The following code creates a button and adds it to the scene object.

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class ButtonTest extends Application {
    @Override
    public void start(Stage primaryStage) throws Exception {
        // TODO Auto-generated method stub
        StackPane root = new StackPane();
        Button btn=new Button("This is a button");
        Scene scene=new Scene(root,300,300);
        root.getChildren().add(btn);
        primaryStage.setScene(scene);
        primaryStage.setTitle("Button Class Example");
        primaryStage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```

Checkbox

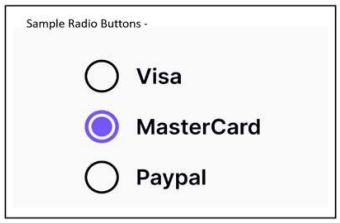


Checkbox is a user interface component that allows the user to select or deselect an option. It is most commonly used to create multiple-choice questions, preferences, filters, and many more. The figure below shows a filter feature with multiple options, allowing users to choose a category and brand according to their preferences. In JavaFX, the checkbox is represented by a class named CheckBox. This class belongs to the javafx.scene.control package. By instantiating this class, we can create a checkbox in JavaFX. Constructors of the CheckBox class are listed below −

- CheckBox() − It is the default constructor that constructs a CheckBox without any option name.
- CheckBox(String str) − It constructs a new CheckBox with the specified option.

The most commonly used constructor of the CheckBox class is its parameterized constructor. It accepts a text representing the option name of the CheckBox. Once the checkbox is created, define a layout pane, such as Vbox or Hbox by passing the CheckBox object to its constructor. Then, create a Scene and pass the object of layout pane as a parameter value to its constructor. Next, set the stage and title of the JavaFX application. Finally, call the main() method to launch the application.

Radiobutton



A button is a component, which performs an action like submit and login when pressed. It is usually labeled with a text or an image specifying the respective action. A radio button is a type of button, which is circular in shape. It has two states, selected and deselected. The figure below shows a set of radio buttons. In JavaFX, the RadioButton class represents a radio button which is a part of the package named javafx.scene.control. It is the subclass of the ToggleButton class. Action is generated whenever a radio button is pressed or released. Generally, radio buttons are grouped using toggle groups, where you can only select one of them. We can set a radio button to a group using the setToggleGroup() method. To create a radio button, use the following constructors −

- RadioButton() − This constructor will create radio button without any label.
- RadioButton(String str) − It is the parameterized constructor which constructs a radio button with the specified label text.

Textfield

The text field is a graphical user interface component used to accept user input in the form of text. It is the most easiest way of interaction between system and user. In JavaFX, the TextField class represents the text field which is a part of the package named javafx.scene.control. Using this we can accept input from the user and read it to our application. This class inherits the TextInputControl which is the base class of all the text controls class. To create a text field, instantiate the TextField class using any of the below constructors −

- TextField() − This constructor will create an empty textfield.
- TextField(String str) − It is the parameterized constructor which constructs a textfield with the specified text.

Textarea

The TextArea control is a graphical user interface component that allow users to enter and display multiple lines of plain text. It is mostly used to collect information like comments, feedbacks and descriptions. In JavaFX, the text area is represented by a class named TextArea which is a part of javafx.scene.control package. By instantiating this class, we can create a text area in JavaFX. Its constructors are listed below −

- TextArea() − It is the default constructor that constructs a text area without any texts.
- TextArea(String str) − It constructs a new text area with the predefined text.

Once the TextArea is created, it can be customized to enhance its appearance and behavior using its properties. For instance, we can set the preferred number of rows and columns using the prefRowCount and prefColumnCount properties respectively. Additionally, we can also enable or disable text wrapping using the wrapText property. The TextArea also supports showing prompt text when there is no text in the component. This is a useful way of informing the user what is expected in the text area, without using tooltips or labels. The prompt text can be set using the setPromptText() method or the promptText property.

Combobox

ComboBox is a part of the JavaFX library. JavaFX ComboBox is an implementation of simple ComboBox which shows a list of items out of which user can select at most one item, it inherits the class ComboBoxBase. Constructors of ComboBox:

- ComboBox(): creates a default empty combo box
- ComboBox(ObservableList i): creates a combo box with the given items
-

Commonly used Methods:

- getEditor(): This method gets the value of the property editor
- getItems(): This method returns the items of the combo box

- getVisibleRowCount(): This method returns the value of the property visibleRowCount.
- setItems(ObservableList v): This method Sets the items of the combo box
- setVisibleRowCount(int v): This method sets the value of the property VisibleRowCount

Listview

The ListView is a graphical user interface component used for displaying a list of items from which a user can select desired items. Generally, a list refers to a group or collection of items. It helps in organizing, structuring and presenting information in more user-friendly and readable way. In JavaFX, the list view is represented by a class named ListView which is a part of javafx.scene.control package. We can create a list view component by instantiating this class. Additionally, we have the option to select its orientation, allowing it to be displayed either vertically or horizontally. List of constructors of the ListView class is as follows −

- ListView() − It is the default constructor that constructs a vertical list view.
- ListView(ObservableList<type> listItems) − It constructs a new vertical list view with the specified list items.

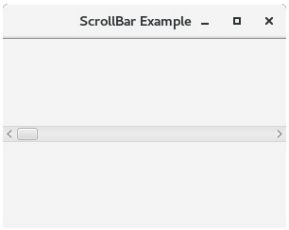
To create a ListView in any JavaFX application, we can use either the default contructor or the parameterized constructor of the ListView class. If the default constructor is used, we should pass the list items explicitly. The parameterized constructor accepts an ArrayList object as a parameter value as shown in the below code block.

```
//list View for educational qualification
ObservableList<String>                    names                    =
FXCollections.observableArrayList("Engineering",      "MCA",      "MBA",
"Graduation", "MTECH", "Mphil", "Phd");
ListView<String> listView = new ListView<String>(names);
```

Once the ListView class is instantiated and its items are added, define any layout pane such VBox or HBox to hold the list view. Next, create a Scene object by passing the layout pane obejct and the dimensions of the Scene to its constructor. Then, set the stage and launch the application to display the result.

Scrollbar

JavaFX Scroll Bar is used to provide a scroll bar to the user so that the user can scroll down the application pages. It can be created by instantiating javafx.scene.control.ScrollBar class. The following code implements scrollbar into our application.




```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.ScrollBar;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;
public class ScrollBar extends Application{
   @Override
   public void start(Stage primaryStage) throws Exception {
       // TODO Auto-generated method stub
       ScrollBar s = new ScrollBar();
       StackPane root = new StackPane();
       root.getChildren().add(s);
       Scene scene = new Scene(root,300,200);
       primaryStage.setScene(scene);
       primaryStage.setTitle("ScrollBar Example");
       primaryStage.show();
   }
   public static void main(String[] args) {
       launch(args);
   }
```

```
}
```

Slider

Slider is a UI control consisting of a track and a thumb along with other optional elements like tick marks and tick labels. It allows the user to select a value from a continuous or discrete range of values by dragging the thumb along a track. The common applications of a slider can be seen in audio or video players, games, brightness controllers and so on. In JavaFX, the slider is represented by a class named Slider. This class belongs to the package javafx.scene.control. By instantiating this class, we can create a slider in JavaFX. Constructors of the Slider class are listed below −

- Slider() − It is the default constructor.
- Slider(double minVal, double maxVal, double currentVal) − It creates a new slider with the specified initial, minimum and maximum values.

To create a Slider in JavaFX, follow the steps given below:

Step 1: Instantiate the Slider class: Instantiate the Slider class inside the start() method. This action will create a slider control in JavaFX application.

```
// creating a slider using default constructor
Slider slide = new Slider();
```

Step 2: Set the minimum and maximum values of Slider: Every slider comes with minimum and maximum values. To set the minimum value, a built-in method named setMin() is used which accepts a double value as an argument. Similarly, the maximum value is set using the setMax() method, which also requires double as a parameter value.

```
// setting the minimum and maximum values
slide.setMin(0);
slide.setMax(100);
```

Step 3: Set the Orientation of the Slider: To set the orientation of the Slider, we use the method named setOrientation(). It accepts enum values VERTICAL or HORIZONTAL as an argument.

```
// setting the orientation
slide.setOrientation(Orientation.VERTICAL);
```

Step 4: Optional Properties of Slider: We can also set the various properties of the Slider such as major and minor tick counts, tick marks and labels. To do so, use the following code −

```
// major and minor tick marks
slide.setMajorTickUnit(10);
slide.setMinorTickCount(9);
slide.setShowTickMarks(true);
slide.setShowTickLabels(true);
```

Step 5: Launch the Application: Once the Slider is created and its properties are set, follow the given steps below to launch the application properly −

- Firstly, create a VBox that holds the slider.
- Next, instantiate the class named Scene by passing the VBox object as a parameter value to its constructor along with the dimensions of the application screen.
- Then, set the title of the stage using the setTitle() method of the Stage class.
- Now, a Scene object is added to the stage using the setScene() method of the class named Stage.
- Display the contents of the scene using the method named show().
- Lastly, the application is launched with the help of the launch() method.

Media with JavaFX

Modern world's rich internet applications must be capable to play and edit the media files when required. JavaFX provides the media-rich API that can play audio and video on the user's demand. JavaFX Media API enables the users to incorporate audio and video into the rich internet applications (RIAs). JavaFX media API can distribute the media content across the different range of devices like TV, Mobile, Tablets and many more. In this part of the tutorial, we will discuss the capability of JavaFX to deal with the media files in an interactive way. For this purpose, JavaFX provides the package javafx.scene.media that contains all the necessary classes. javafx.scene.media contains the following classes.

- javafx.scene.media.Media
- javafx.scene.media.MediaPlayer
- javafx.scene.media.MediaStatus
- javafx.scene.media.MediaView

Media Events: The JavaFX team have designed media API to be event driven. The callback behaviour attached with the media functions are used to handle media events. Instead of typing code for a button via a EventHandler, a code is implemented that responds to the triggering of the media player's OnXXXX events where XXXX is the event name. java.lang.Runnable functional interfaces are used as the callbacks which are invoked when an event is encountered. When playing the media content in javafx, we would create the Lambda expressions (java.lang.Runnable interfaces) to be set on the onReady event. Consider the following example.

```
Media media = new Media(url);
MediaPlayer mediaPlayer = new MediaPlayer(media);
Runnable playMusic = () -> mediaPlayer.play();
mediaPlayer.setOnReady(playMusic);
```

The playMusic variable is assigned to a lambda expression. This get passed into the Media player's setOnReady() method. The Lambda expression will get invoked when the onReady event is encountered.

Playing Audio: We can load the audio files with extensions like .mp3,.wav and .aifff by using JavaFX Media API. We can also play the audio in HTTP live streaming format. It is the new feature introduced in JavaFX 8 which is also known as HLS. Playing audio files in JavaFX is simple. For this purpose, we need to instantiate javafx.scene.media.Media class by passing the audio file path in its constructor. The steps required to be followed in order to play audio files are described below.

- Instantiate the javafx.scene.media.Media class by passing the location of the audio file in its constructor. Use the following line of code for this purpose.

  ```
  Media media = new Media("http://path/file_name.mp3");
  ```

- Pass the Media class object to the new instance of javafx.scene.media.MediaPlayer object.

  ```
  Mediaplayer mediaPlayer = new MediaPlayer(media);
  ```

- Invoke the MediaPlayer object's play() method when onReady event is triggered.

  ```
  MediaPlayer.setAutoPlay(true);
  ```

The Media File can be located on a web server or on the local file system. SetAutoPlay() method is the short-cut for setting the setOnReady() event handler with the lambda expression to handle the event.

Playing Video: Playing video in JavaFX is quite simple. We need to use the same API as we have used in the case of playing Audio files. In the case of playing video, we need to use the MediaView node to display the video onto the scene. For this purpose, we need to instantiate the MediaView class by passing the Mediaplayer object into its constructor. Due to the fact that, MediaView is a JavaFX node, we will be able to apply effects

to it. In this part of the tutorial, we will discuss the steps involved in playing video media files and some examples regarding this. Steps to play video files in JavaFX:

- Instantiate the javafx.scene.media.Media class by passing the location of the audio file in its constructor. Use the following line of code for this purpose.

```
Media media = new Media("http://path/file_name.mp3");
```

- Pass the Media class object to the new instance of javafx.scene.media.MediaPlayer object.

```
Mediaplayer mediaPlayer = new MediaPlayer(media);
```

- Invoke the MediaPlayer object's play() method when onReady event is triggered.

```
mediaPlayer.setAutoPlay(true);
```

- Instantiate MediaView class and pass Mediaplayer object into its constructor.

```
MediaView mediaView = new MediaView (mediaPlayer);
```

- Add the MediaView Node to the Group and configure Scene.

```
Group root = new Group();
root.getChildren().add(mediaView)
Scene scene = new Scene(root,600,400);
primaryStage.setTitle("Playing Video");
primaryStage.show();
```