Object Oriented Programming and JAVA Language

Object Oriented Programming and JAVA Language

- Object Oriented Programming with Core Java
- Java Programming features
- JVM, Byte codes and Class path
- Java Program Development
- Compilation and Execution of JAVA programs
- Basic JAVA language elements – keywords, comments, data types and variables.
- JAVA Arithmetic, Assignment, Relational, Logical, Increment / Decrement operators and expressions.
- JAVA String Operators
- JAVA Input and Output streams, System in, System out.
- Input using Scanner class and Console class methods

OBJECT ORIENTED PROGRAMMING WITH CORE JAVA

Object-Oriented Programming is a paradigm that provides many concepts, such as inheritance, data binding, polymorphism, etc. The programming paradigm where everything is represented as an object is known as a truly object-oriented programming language.

Object-oriented programming has several advantages over procedural programming:

- OOP is faster and easier to execute
- OOP provides a clear structure for the programs
- OOP helps to keep the Java code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter development time.

What is OOPS in Java?

OOPs can be defined as:

A mo dular approach where data and functions can be combined into a single unit known as an object. It emphasises data and security and provides the reusability of code. Using OOP, we can resemble our code in the real world.

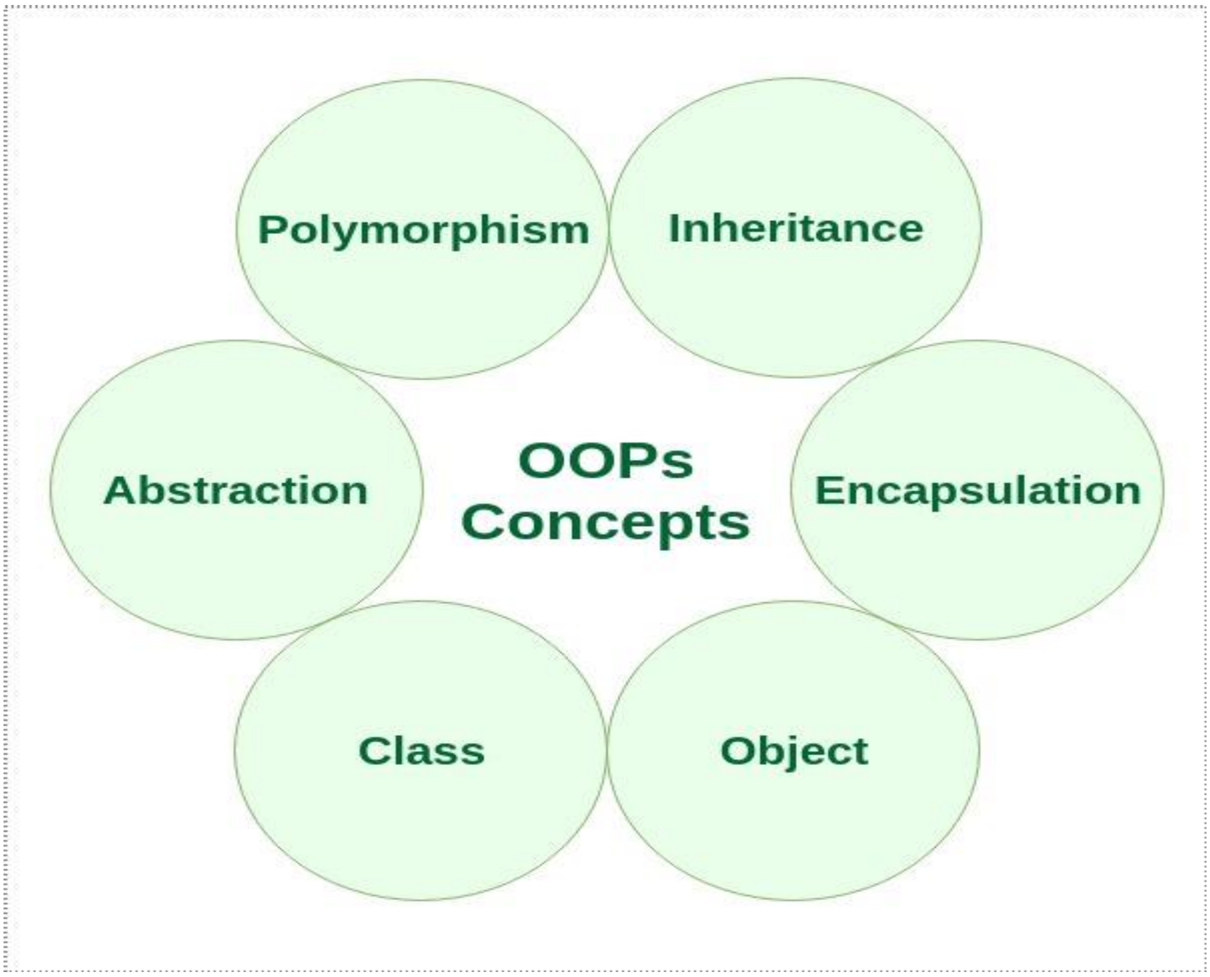
List of Java OOPs Concepts

The following are the concepts in OOPs :

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Apart from these concepts, there are some other terms which are used in Object-Oriented design:

- Coupling
- Cohesion
- Association
- Aggregation
- Composition



Object:

Any entity that has state and behaviour 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.

Class:

A class can be considered as a blueprint using which you can create as many objects as you like. The class is a group of similar entities. It is only a logical component and not a physical entity. **For example,** if you had a class called "Expensive Cars" it could have objects like Mercedes, BMW, Toyota, etc.

Inheritance:

Inheritance is an OOPS concept in which one object acquires the properties and behaviours of the parent object. It's creating a parent-child relationship between two classes. It offers a robust and natural mechanism for organising and structuring any software. It provides code reusability. It is used to achieve **runtime polymorphism**. **For example,** let's say we have an Employee class. Employee class has all common attributes and methods which all employees must have within the organisation. There can be other specialised employees as well e.g. Manager. Managers are regular employees of the organisation but, additionally, they have few more attributes over other employees e.g. they have reports or subordinates.

Polymorphism

It refers to the ability of object-oriented programming languages to differentiate between entities with the same name efficiently. This is done by Java with the help of the signature and declaration of these entities. The ability to appear in many forms is called **polymorphism**.

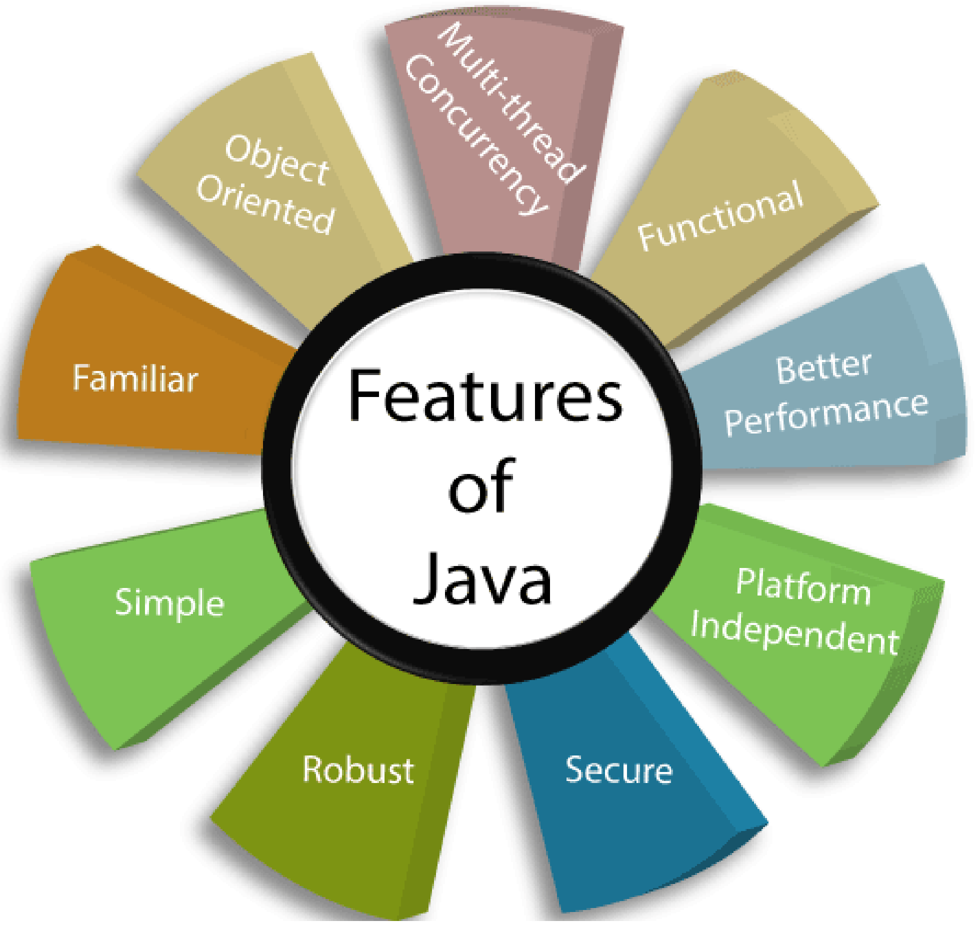
Abstraction

Abstraction means showing only the relevant details to the end-user and hiding the irrelevant features that serve as a distraction. For example, during an ATM operation, we only answer a series of questions to process the transaction without any knowledge about what happens in the background between the bank and the ATM.

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.

Features of JAVA Programming:



Java is simple

Java is Easy to write and more readable and eye

Java has a concise, cohesive set of features that makes it easy to learn and use.

Most of the concepts are drawn from C++ thus making Java learning simpler.

Java is secure

Java programs can not harm other systems thus secure. Java provides a secure means of creating Internet applications. Java provides a secure way to access web applications.

Java is portable

Java programs can execute in any environment which there is a Java run-time system.(JVM) Java programs can be run on any platform (Linux,Windows,Mac) for Java programs can be transferred over world wide web (e.g applets).

Java is object-oriented

Java programming is an object-oriented programming language.Like C++ java provides most of the object oriented features. Java is pure OOP. Language. (while C++ is semi object oriented)

Java is robust

Java encourages error-free programming by being strictly typed and performing run-time checks.

Java is multithreaded

Java provides integrated support for multithreaded programming.

Java is architecture-neutral

Java is architecture-neutral

Java is not tied to a specific machine or operating system architecture. Machine Independent i.e Java is independent of hardware.

Java is interpreted

Java supports cross-platform code through the Java bytecode. use of Bytecode can be interpreted on any platform by JVM.

Java is distributed

Java was designed with the distributed environment. Java can be transmit,run over the internet.

JVM ,Byte codes and Class path

JDK, JRE, and JVM are essential components in the world of Java programming, each serving a specific role in the development and execution of Java applications:

JDK (Java Development Kit)

The JDK is a software package provided by Oracle (and other vendors) that includes all the tools and libraries necessary for Java application development. It contains the following key components:

- **Java Compiler (javac):** This tool is used to compile Java source code (.java files) into bytecode (.class files), which can be executed by the Java Virtual Machine (JVM).
- **Java Runtime Environment (JRE):** The JRE is also included in the JDK. It is needed for running Java applications but not for developing them. The JRE consists of the JVM and essential libraries.
- **Development Tools:** The JDK includes various development tools like debugging tools, JavaDoc for generating documentation, and more.

Developers use the JDK to write, compile, and package Java applications.

JRE (Java Runtime Environment):

The JRE is a subset of the JDK and is required to run Java applications. It includes the following components:

- **Java Virtual Machine (JVM):** The JVM is the runtime engine responsible for executing Java bytecode. It interprets and translates bytecode into machine-specific code that the underlying operating system can understand. Each platform (e.g., Windows, Linux, macOS) has its own JVM implementation,

- **Java Standard Library:** The JRE includes a set of core libraries and classes that provide essential functionality to Java applications. These libraries offer features like input/output operations, networking, and more.

End-users who want to run Java applications need the JRE installed on their system. It allows them to execute Java programs without the need for development tools.

JVM (Java Virtual Machine):

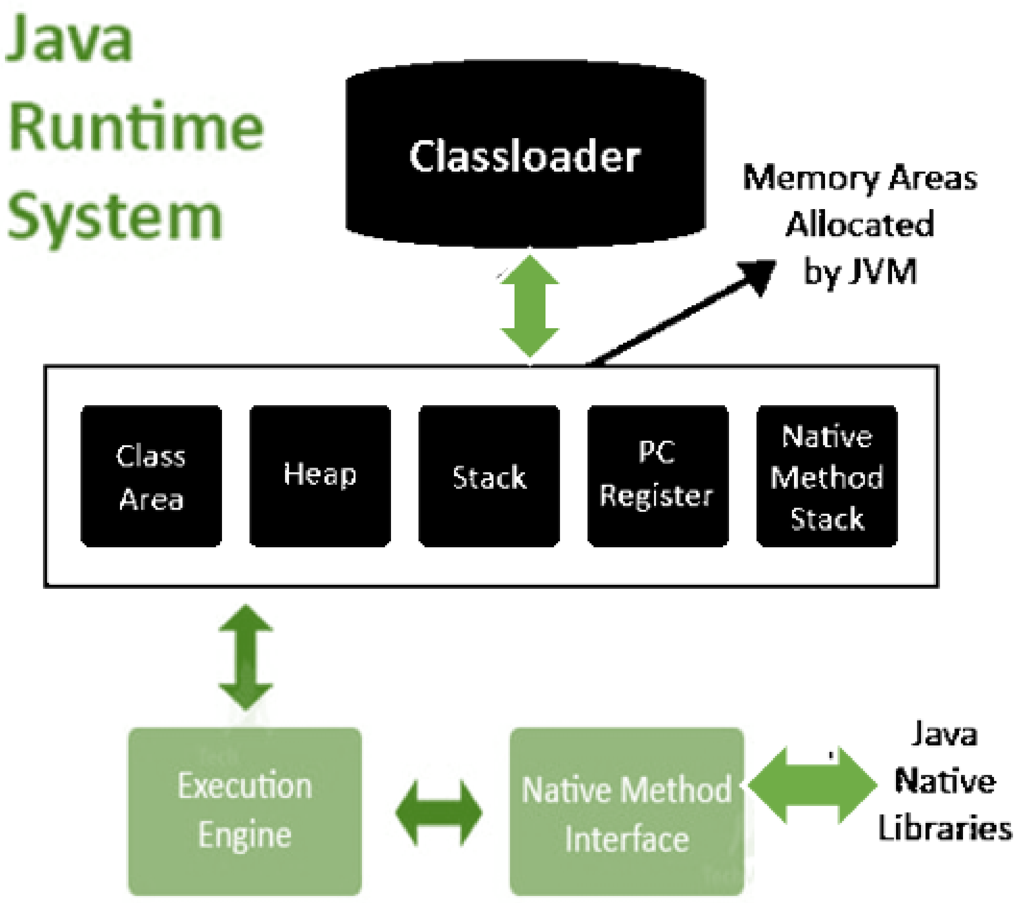
The JVM is the runtime environment in which Java bytecode is executed. It is an integral part of both the JDK and the JRE. The JVM performs several crucial tasks:

- **Loading:** It loads compiled Java bytecode (class files) into memory.
- **Verification:** The JVM checks the bytecode for security and integrity to prevent potential security vulnerabilities.
- **Execution:** The bytecode is executed by the JVM, which translates it into machine code specific to the underlying hardware and operating system.
- **Memory Management:** The JVM manages memory allocation and garbage collection to ensure efficient memory usage.
- **Security:** The JVM enforces Java's security model, including access control, class-loading restrictions, and more.

JVM Architecture

Java applications are often referred to as WORA, which stands for "Write Once Run Anywhere." This concept signifies that a programmer can write Java code on one system and anticipate it to run on any other Java-enabled system without the need for modifications. This seamless portability is made possible due to the presence of the Java Virtual Machine (JVM).

When a .java file is compiled, it produces **.class** files that contain bytecode and share the same class names as the corresponding **.java** files. When these **.class** files are executed, they undergo a series of steps that collectively define the functionality of the JVM (Java Virtual Machine).



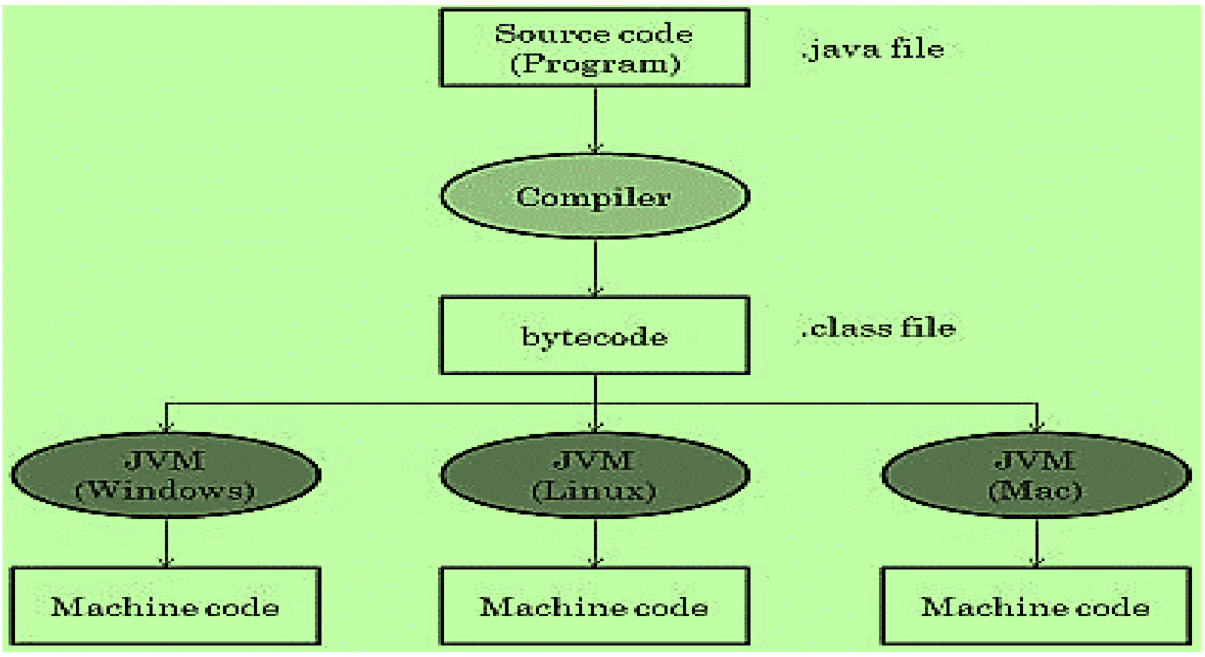
Java bytecode

Java bytecode serves as the instruction set for the Java Virtual Machine (JVM). It functions akin to an assembler, providing a symbolic representation of Java code, much like how C++ code can be represented in an alias form. When a Java program is compiled, it results in the generation of Java bytecode. To put it succinctly, Java bytecode can be thought of as machine code encapsulated in a .class file. The utilisation of Java bytecode is what enables Java to attain platform independence, ensuring that Java programs can run on different systems without modification.

How does it work?

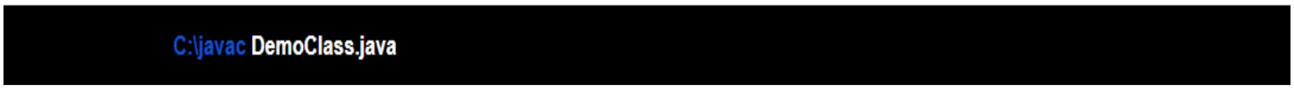
When we develop a program in Java, the initial step involves compiling the code, resulting in the creation of bytecode. This bytecode serves as an intermediary representation of the code. Importantly, when we intend to execute this .class file on different platforms, we can readily do so. Following the initial compilation, it's the Java Virtual Machine (JVM), rather than the specific processor of the platform, that takes charge of running the bytecode.

In essence, this implies that we only need a basic Java installation on any platform where we wish to execute our code. The JVM plays a crucial role by managing the resources required for executing the bytecode. It communicates with the processor to allocate the necessary resources. It's worth noting that JVMs operate in a stack-based manner, using a stack implementation to interpret and execute the bytecode instructions.



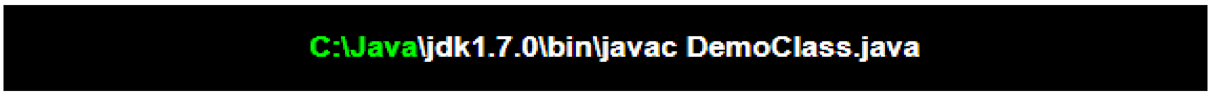
What is the path ?

After installing Java on your computer, it's important to configure the PATH environment variable. This configuration allows you to conveniently execute Java-related executables (such as javac.exe, java.exe, javadoc.exe, etc.) from any directory without the need to specify the full path to the command.



```
C:\javac DemoClass.java
```

If you don't set the PATH environment variable, you would indeed have to specify the full path each time you run a Java command, like this:

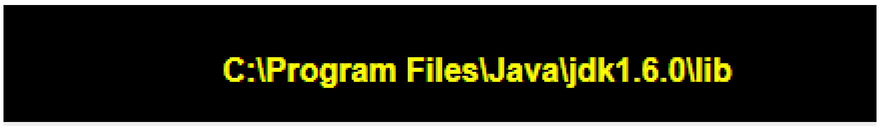


```
C:\Java\jdk1.7.0\bin\javac DemoClass.java
```

Without the PATH variable configuration, this full path specification is necessary for running Java executables.

What is Classpath?
The classpath is a system environment variable utilised by both the Java compiler and the JVM (Java Virtual Machine).

It serves as a reference to help these components locate the necessary class files.



```
C:\Program Files\Java\jdk1.6.0\lib
```

Java Program Development Compilation and Execution of JAVA programs

Developing, compiling, and executing Java programs involves several steps. Java is a popular programming language that follows a "write once, run anywhere" philosophy, meaning you can write code on one platform and run it on any platform that has a Java Virtual Machine (JVM) installed. Here's a step-by-step guide:

1.Install Java Development Kit (JDK): To develop Java programs, you need to have the Java Development Kit (JDK) installed on your computer. You can download the JDK from the official Oracle website or choose an open-source alternative like OpenJDK. Make sure to set up the PATH environment variable to include the JDK's bin directory.

2.Choose a Text Editor or Integrated Development Environment (IDE): You can write Java code using a simple text editor like Notepad or a more sophisticated Integrated Development Environment (IDE) like Eclipse, IntelliJ IDEA, or Visual Studio Code. IDEs often provide features like code completion, debugging, and project management.

3.Write Your Java Code: Create a new Java source code file with a .java extension. For example, you can create a file named MyProgram.java. Write your Java code in this file. Here's a simple "Hello, World!" program in Java:

```
public class MyProgram {
    public static void main(String[] args) {
      System.out.println("Hello, World!");
      }
  }
```

4.Save Your Java File: Save your Java source code file in a directory of your choice.

5.Compile Your Java Code: Open your terminal or command prompt and navigate to the directory where you saved your Java source file. To compile the code, use the **javac** command followed by your Java source file's name:

```
javac MyProgram.java
```

If there are no errors in your code, this will generate a bytecode file with a **.class**' extension, such as '**MyProgram.class'.**

6.Execute Your Java Program: After successfully compiling your Java code, you can execute it using the '**java**' command followed by the class name (excluding the '**.class**' extension):

```
java MyProgram
```

If everything is correct, you should see the output of your program in the terminal:

```
Hello, World!
```

7.Debugging and Testing: Use debugging tools provided by your chosen IDE or standard tools like '**System.out.println()**' to debug and test your Java code.

8.Packaging and Distribution (if necessary): If you want to distribute your Java program, you can package it into a JAR (Java Archive) file, which contains all the necessary class files and resources. This allows others to run your program easily.

Java Program Development Compilation and Execution of JAVA programs

Developing, compiling, and executing Java programs involves several steps. Java is a popular programming language that follows a "write once, run anywhere" philosophy, meaning you can write code on one platform and run it on any platform that has a Java Virtual Machine (JVM) installed. Here's a step-by-step guide:

1.Install Java Development Kit (JDK): To develop Java programs, you need to have the Java Development Kit (JDK) installed on your computer. You can download the JDK from the official Oracle website or choose an open-source alternative like OpenJDK. Make sure to set up the PATH environment variable to include the JDK's bin directory.

2.Choose a Text Editor or Integrated Development Environment (IDE): You can write Java code using a simple text editor like Notepad or a more sophisticated Integrated Development Environment (IDE) like Eclipse, IntelliJ IDEA, or Visual Studio Code. IDEs often provide features like code completion, debugging, and project management.

3.Write Your Java Code: Create a new Java source code file with a .java extension. For example, you can create a file named MyProgram.java. Write your Java code in this file. Here's a simple "Hello, World!" program in Java:

```
public class MyProgram {
    public static void main(String[] args) {
      System.out.println("Hello, World!");
      }
  }
```

4.Save Your Java File: Save your Java source code file in a directory of your choice.

5.Compile Your Java Code: Open your terminal or command prompt and navigate to the directory where you saved your Java source file. To compile the code, use the **javac** command followed by your Java source file's name:

```
javac MyProgram.java
```

If there are no errors in your code, this will generate a bytecode file with'.class extension, such as **'MyProgram.class'.**

6.Execute Your Java Program: After successfully compiling your Java code, you can execute it using the **'java'** command followed by the class name (excluding the **'.class'** extension):

```
java MyProgram
```

If everything is correct, you should see the output of your program in the terminal:

```
Hello, World!
```

7.Debugging and Testing: Use debugging tools provided by your chosen IDE or standard tools like **'System.out.println()'** to debug and test your Java code.

8.Packaging and Distribution (if necessary): If you want to distribute your Java program, you can package it into a JAR (Java Archive) file, which contains all the necessary class files and resources. This allows others to run your program easily.

Basic JAVA language elements keywords, comments, data types and variables

KEYWORDS

There are certain words with a specific meaning in java which tell (help) the compiler what the program is supposed to do. These Keywords cannot be used as variable names, class names, or method names. Keywords in java are case sensitive, all characters being lower case.

Keywords are reserved words that are predefined in the language; see the table below (Taken from Sun Java Site). All the keywords are in lowercase.

abstract	default	if	private	this
boolean	do	implements	protected	throw
break	double	import	public	throws
byte	else	instanceof	return	transient
case	extends	int	short	try
catch	final	interface	static	void
char	finally	long	strictfp	volatile
class	float	native	super	while
const	for	new	switch	
continue	goto	package	synchronized	

COMMENTS

Comments are descriptions that are added to a program to make code easier to understand. The compiler ignores comments and hence its only for documentation of the program.

Java supports three comment styles.

Block style comments begin with /* and terminate with */ that spans multiple lines.

Line style comments begin with // and terminate at the end of the line. (Shown in the above program)

Documentation style comments begin with /** and terminate with */ that spans multiple lines. They are generally created using the automatic documentation generation tool, such as javadoc. (Shown in the above program)

Data Types in Java

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

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

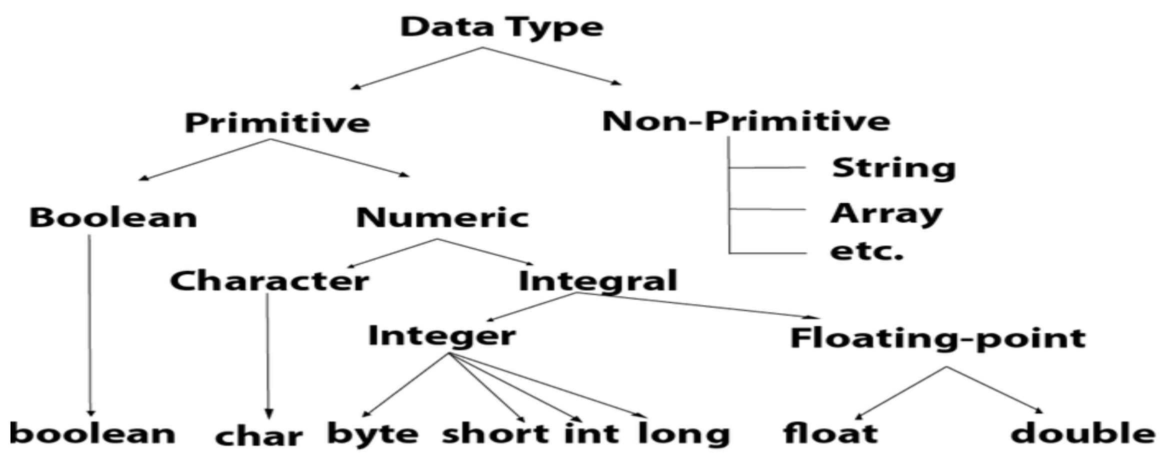
Java Primitive Data Types

In Java language, primitive data types are the building blocks of data manipulation. These are the most basic data types available in Java language.

here are 8 types of primitive data types:

- boolean data type
- byte data type
- char data type
- short data type
- int data type
- long data type

- float data type
- double data type



What is a Variable in Java?

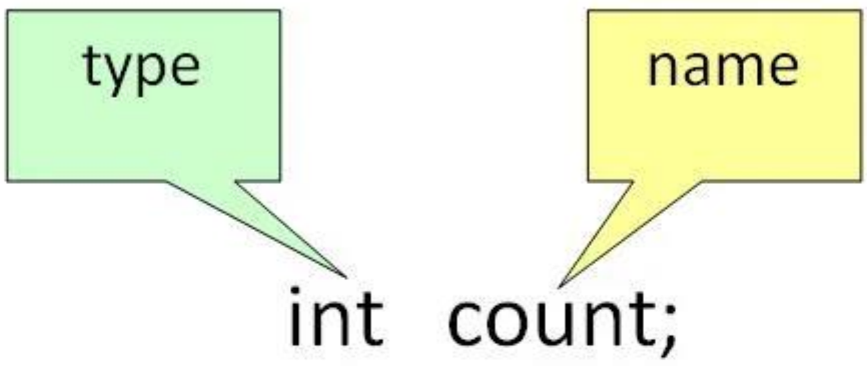
Variable in Java is a data container that stores the data values during Java program execution. Every variable is assigned a data type which designates the type and quantity of value it can hold. Variable is a memory location name of the data. The Java variables have mainly three types : Local, Instance and Static.

In order to use a variable in a program you to need to perform 2 steps

1. Variable Declaration
2. Variable Initialization

Variable Declaration:

To declare a variable, you must specify the data type & give the variable a unique name.



Examples of other Valid Declarations are

```
int a,b,c;

float pi;

double d;

char a;
```

JAVA Arithmetic, Assignment, Relational, Logical, Increment / Decrement operators and expressions.

Java Operators:

Operators are used to perform operations on variables and values.

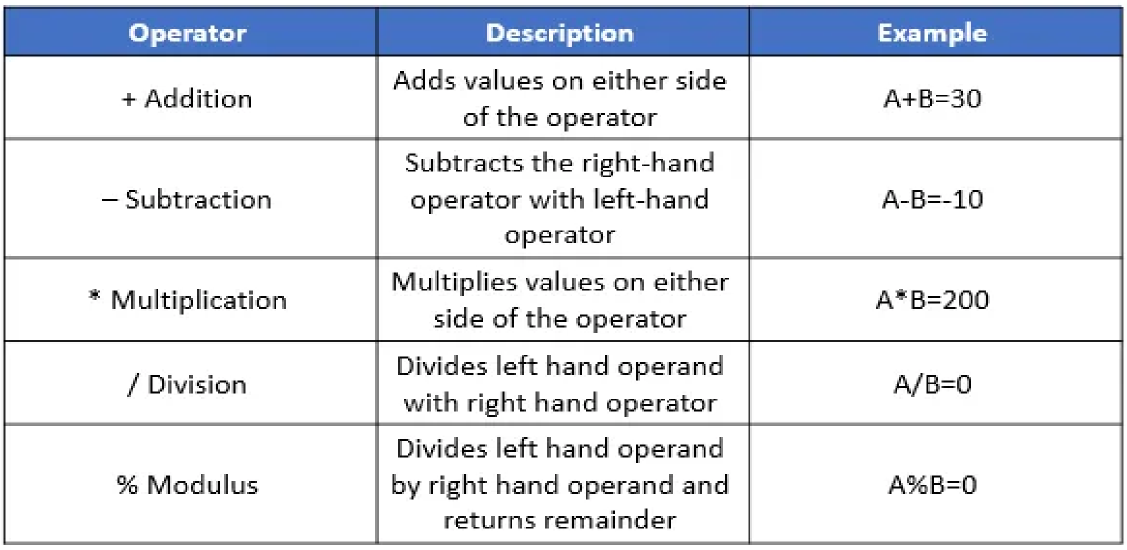
Java supports the following types of operators:

- Arithmetic Operators
- Assignment Operators
- Logical Operators
- Relational Operators
- Unary Operators
- Bitwise Operators

- Ternary Operators
- Shift Operators

Arithmetic Operators in Java

Arithmetic Operators are used to performing mathematical operations like addition, subtraction, etc. Assume that A = 10 and B = 20 for the below table.



Operator	Description	Example
+ Addition	Adds values on either side of the operator	A+B=30
– Subtraction	Subtracts the right-hand operator with left-hand operator	A-B=-10
* Multiplication	Multiplies values on either side of the operator	A*B=200
/ Division	Divides left hand operand with right hand operator	A/B=0
% Modulus	Divides left hand operand by right hand operand and returns remainder	A%B=0

```java
// Java code to illustrate Addition operator

import java.io.*;

class Addition {
    public static void main(String[] args)
    {
        // initializing variables
        int num1 = 10, num2 = 20, sum = 0;

        // Displaying num1 and num2
        System.out.println("num1 = " + num1);
        System.out.println("num2 = " + num2);

        // adding num1 and num2
        sum = num1 + num2;
        System.out.println("The sum = " + sum);
    }
}
```

Output **:** num1 = 10 num2 = 20

The sum = 30

Assignment Operators:

- **= (Assignment):** Assigns the value of the right operand to the left operand.
- **+= (Addition Assignment):** Adds the right operand to the left operand and assigns the result to the left operand.
- **-= (Subtraction Assignment)**: Subtracts the right operand from the left operand and assigns the result to the left operand.
- ***= (Multiplication Assignment):** Multiplies the left operand by the right operand and assigns the result to the left operand.

- **/= (Division Assignment):** Divides the left operand by the right operand and assigns the result to the left operand.
- **%= (Modulus Assignment):** Computes the modulus of the left operand and the right operand and assigns the result to the left operand.

Example: int x = 10;

```
 x += 5; // x is now 15
 x -= 3; // x is now 12
 x *= 2; // x is now 24
 x /= 4; // x is now 6
 x %= 3; // x is now 0
```

Relational Operators:

- == (Equal to): Checks if two operands are equal.
- != (Not equal to): Checks if two operands are not equal.
- < (Less than): Checks if the left operand is less than the right operand.
- > (Greater than): Checks if the left operand is greater than the right operand.
- <= (Less than or equal to): Checks if the left operand is less than or equal to the right operand.
- >= (Greater than or equal to): Checks if the left operand is greater than or equal to the right operand.

Example:

```
int num1 = 10, num2 = 20;
boolean isEqual = (num1 == num2); // false
boolean isNotEqual = (num1 != num2); // true
boolean isLessThan = (num1 < num2); // true
boolean isGreaterThan = (num1 > num2); // false
```

Logical Operators:

- && (Logical AND): Returns true if both operands are true.
- || (Logical OR): Returns true if at least one operand is true.
- ! (Logical NOT): Returns the opposite boolean value of the operand.

Example:

```
boolean isTrue = true, isFalse = false;

boolean result1 = isTrue && isFalse; // false

boolean result2 = isTrue || isFalse; // true

boolean result3 = !isTrue; // false
```

Increment and Decrement Operators:

- ++ (Increment): Increases the value of the operand by 1.
- -- (Decrement): Decreases the value of the operand by 1.

Example:1

```
int count = 5;

count++; // count is now 6

count--; // count is now 5
```

Example:2

```
int j = 0, k = 0;  // Initially both j and k are 0

j = ++k;  // Final values of both j and k are 1
```

Example:3

```
int i = 0, k = 0;     // Initially both i and k are 0
i = k++;                      // Final value of i is 0 and k is
```

Operator	Description
()	Parentheses (grouping)
[]	Array subscript
.	Member access (dot operator)
++ and --	Post-increment and post-decrement
+ (unary) and - (unary)	Unary plus and unary minus
! and ~	Logical NOT and bitwise NOT
*, /, and %	Multiplication, division, and modulo
+ and -	Addition and subtraction
<< and >>	Bitwise left shift and bitwise right shift
<, <=, >, and >=	Relational operators (less than, less/equal, greater than, greater/equal)
instanceof	Type comparison (checks if an object is an instance of a class or interface)
== and !=	Equality and inequality (equals and not equals)
&	Bitwise AND
^	Bitwise XOR
`	`
&&	Conditional AND (short-circuiting)
`	
? :	Ternary conditional (conditional operator)
=	Assignment
+=, -=, *=, /=, %=	Assignment with addition, subtraction, multiplication, division, and modulo
&=, ^=, `	=`
<<=, >>=, >>>=	Assignment with bitwise left shift, right shift, and unsigned right shift

Examples of Operator:

```
public class OperatorDemo {
public static void main(String[] args) {
// Arithmetic operators
int num1 = 10;
int num2 = 5;
int sum = num1 + num2;
int difference = num1 - num2;
int product = num1 * num2;
int quotient = num1 / num2;
int remainder = num1 % num2;
System.out.println("Arithmetic Operators:");
System.out.println("Sum: " + sum);
System.out.println("Difference: " + difference);
System.out.println("Product: " + product);
System.out.println("Quotient: " + quotient);
System.out.println("Remainder: " + remainder);

 // Assignment operators
int x = 5;
x += 3; // Equivalent to x = x + 3
System.out.println("\nAssignment Operators:");
System.out.println("x after assignment: " + x);

// Relational operators
int a = 10; int b = 20;
boolean isEqual = (a == b);
boolean isNotEqual = (a != b);
boolean isGreater = (a > b);
boolean isLess = (a < b);
System.out.println("\nRelational Operators:");
System.out.println("Is equal: " + isEqual);
System.out.println("Is not equal: " + isNotEqual);
System.out.println("Is greater: " + isGreater);
System.out.println("Is less: " + isLess);

// Logical operators
boolean p = true;
boolean q = false;
boolean logicalAnd = p && q;
boolean logicalOr = p || q;
boolean logicalNot = !p;
System.out.println("\nLogical Operators:");
System.out.println("Logical AND: " + logicalAnd);
System.out.println("Logical OR: " + logicalOr);
System.out.println("Logical NOT: " + logicalNot);

// Increment and Decrement operators
int count = 5; count++; // Increment by 1
int anotherCount = 10;
anotherCount--; // Decrement by 1
System.out.println("\nIncrement/Decrement Operators:");
System.out.println("count after increment: " + count);
System.out.println("anotherCount after decrement: " + anotherCount);
   }
}
```

Output

```
Arithmetic Operators:
Sum: 15
Difference: 5
Product: 50
Quotient: 2
Remainder: 0

Assignment Operators:
x after assignment: 8

Relational Operators:
```

```
Is equal: false
Is not equal: true
Is greater: false
Is less: true
Logical Operators:
Logical AND: false
Logical OR: true
Logical NOT: false
Increment/Decrement Operators:
count after increment: 6
anotherCount after decrement: 9
```

JAVA String Operators

In Java, there aren't specific "string operators" like you might find in languages like JavaScript or Python. Instead, Java primarily relies on methods and concatenation to perform operations on strings. Here are some common string operations and techniques in Java:

For Example:-

1.Concatenation Operator +:

- You can use the + operator to concatenate (join) strings together.

```
String firstName = "John";
String lastName = "Doe";
String fullName = firstName + " " + lastName;
```

Example:-

```
class Concat_Operator {
   public static void main( String args[] ) {
      String first = "Hello";
      String second = "World";

      String third = first + second;
      System.out.println(third);

      // yet another way to concatenate strings
```

```
        first += second;
        System.out.println(first);
    }
}
```

Output

```
HelloWorld
HelloWorld
```

2.Concatenation with Other Data Types:

- You can also use the + operator to concatenate strings with other data types, which automatically converts the non-string operands to strings.
int age = 30;

String message = "My age is " + ag

Example:-

```
public class Demo {
   public static void main(String args[]){
      String st1 = "Hello";
      int age = 500;
      String res = st1+age;
      System.out.println(res);
   }
}
```

Output

```
Hello500
```

3.String Methods:

- Java provides a wide range of methods for working with strings, such as **length(), charAt(), substring(), indexOf(), toUpperCase(), toLowerCase()**, and many others. These methods allow you to perform various operations on strings.
String text = "Hello, World!";
int length = text.length(); // 13

```
char firstChar = text.charAt(0); // 'H'
String substring = text.substring(7); // "World!"
```

4.String Comparison:

- You can compare strings in Java using the **equals()** method for content-based comparison and == for reference-based comparison. It's important to use **equals()** when comparing the contents of two strings because == checks if two string references point to the same memory location.

```
String str1 = "Hello";

String str2 = "Hello";

boolean areEqual = str1.equals(str2); // true
```

Example:-

```
class Test {

 public static void main(String args[]){

   String s1="Rachin";

   String s2="Rachin";

   String s3=new String("Rachin");


//true (because both refer to same instance)

   System.out.println(s1==s2);

//false(because s3 refers to instance created in nonpool)

   System.out.println(s1==s3);

 }

}
```

Output

```
true
false
```

5.String Formatting:

- Java provides the **String.format()** method and the **printf()** method for formatting strings using placeholders and format specifiers, similar to C's **printf().**

```
String formatted = String.format("Hello, %s!", "John");

System.out.println(formatted); // Hello, John!
```

Example:-

```
public class FormatExample{

public static void main(String args[]){

String name="Pradyumn";

String sf1=String.format("name is %s",name);

String sf2=String.format("value is %f",32.33434);


//returns 12 char fractional part filling with 0



System.out.println(sf1);

System.out.println(sf2);

}

}
```

Output

```
name is Pradyumn
value is 32.334340
```

6.StringBuilder and StringBuffer:

- For efficient string manipulation, especially when you need to concatenate strings in a loop or modify them frequently, you can use the StringBuilder (not thread-safe) or StringBuffer (thread-safe) classes.

```
StringBuilder stringBuilder = new StringBuilder();

stringBuilder.append("Hello, ");

stringBuilder.append("World!");

String result = stringBuilder.toString();
```

Example:-

```
public class BufferTest{

    public static void main(String[] args){
        StringBuffer buffer=new StringBuffer("hello");

        buffer.append("java");

        System.out.println(buffer);

    }

}
```

Output

```
hellojava
```

Example:-

```
public class BuilderTest{

    public static void main(String[] args){

        StringBuilder builder=new StringBuilder("hello");

        builder.append("Pradyumn");

        System.out.println(builder);

    }

}
```

Output

Indeed, the Java String class offers a wide range of methods for performing various operations on strings. Some of the commonly used String methods include:

- **compare():** Compares two strings lexicographically.
- **concat():** Concatenates one string with another.
- **equals():** Compares two strings for equality.
- **split():** Splits a string into an array of substrings based on a delimiter.
- **length():** Returns the length (number of characters) of a string.
- **replace():** Replaces occurrences of a specified substring with another substring.
- **compareTo():** Compares two strings lexicographically and returns an integer.
- **intern():** Returns a canonical representation of the string.
- **substring()**: Extracts a portion of the string based on given indices etc.

There are two ways to create String object:

1. By string literal
2. By new keyword
   <String_Type> <string_variable> = "<sequence_of_string>";

   1. String literal

   One of the advantages of using the `intern()` method in Java is to enhance memory efficiency. This is achieved by preventing the creation of new objects if an identical string already exists in the string constant pool.

   Example:
   String demoString = “ctiworld”;

   =

   In the statement **String s = new String("Welcome");**, the JVM operates as follows:

   It creates a new string object in the regular (non-pool) heap memory, and the literal "Welcome" is placed in the string constant pool. Consequently, the variable s will point to the object in the heap (non-pool) memory.
   Example:

   String demoString = new String (“ctiworld”);

   Interfaces and Classes in Strings in Java

   CharBuffer is a class that implements the CharSequence interface. Its primary purpose is to enable the substitution of character buffers for CharSequences. An illustrative application of this capability can be found in the java.util.regex package, where CharBuffer is used.

   A String is essentially a sequence of characters. In Java, String objects are immutable, signifying that they are constant and cannot be altered after their creation

   CharSequence Interface: The CharSequence interface in Java serves as a representation for sequences of characters. Below are some of the classes that implement the CharSequence interface:

1. String
2. StringBuffer
3. StringBuilder

   1. StringBuffer:- StringBuffer is a companion class to String in Java, and it offers extensive functionality for manipulating character sequences. While strings in Java are immutable and represent fixed-length character sequences, StringBuffer represents character sequences that can dynamically grow and be modified.

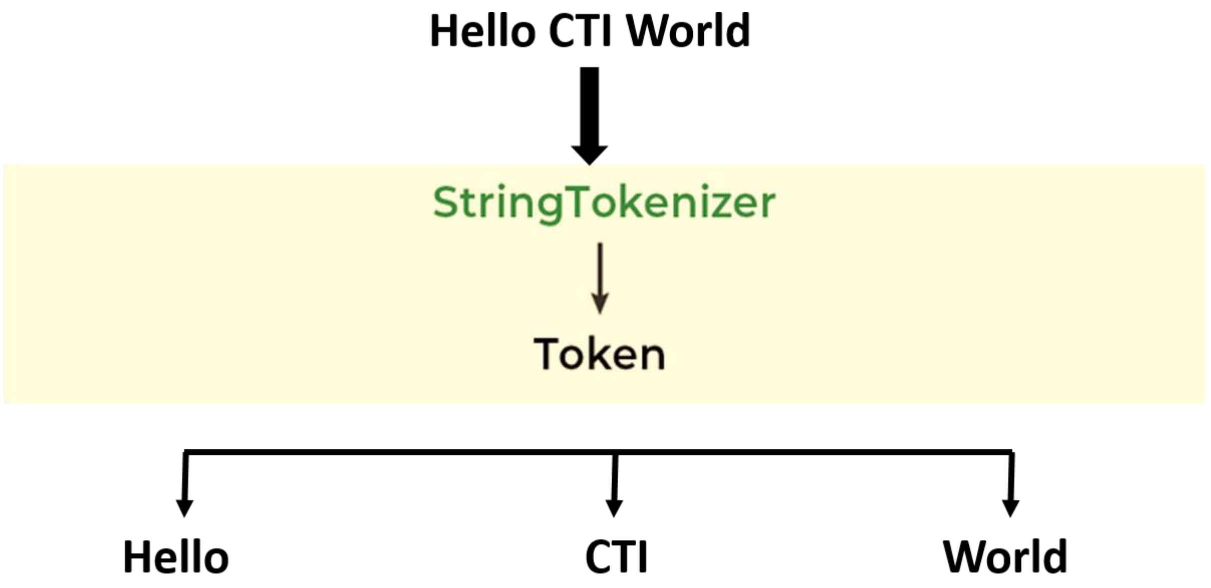
StringBuffer demoString = new StringBuffer("ctiworld");

2. StringBuilder:-In Java, the StringBuilder class represents a mutable sequence of characters. Unlike the String class, which creates immutable character sequences, StringBuilder provides an alternative that allows you to create and modify mutable character sequences

StringBuilder demoString = new StringBuilder();
demoString.append("CTI");

3. StringTokenizer: The StringTokenizer class in Java is employed for splitting a string into tokens or smaller components.

Example:-



Indeed, a StringTokenizer object in Java keeps track of an internal current position within the string that it's tokenizing. Certain operations cause this position to move forward as characters are processed. To obtain a token, the StringTokenizer object extracts a substring from the original string used to create it.

StringJoiner is a class within the java.util package in Java, designed for constructing a sequence of characters (strings) separated by a specified delimiter. It also offers the option to begin with a provided prefix and end with a supplied suffix. While similar functionality can be achieved using the StringBuilder class by manually appending a delimiter after each string, StringJoiner simplifies this process, reducing the amount of code you need to write.
Syntax:
public StringJoiner(CharSequence delimiter)

As mentioned earlier, you can create a string in Java using a string literal. A string literal is a sequence of characters enclosed within double quotation marks

String myString = "Hello, World!";

You're absolutely correct! In Java, when you create a string using a string literal, the JVM checks the String Constant Pool. If an identical string exists in the pool, it returns a reference to that existing string instead of creating a new object. This behaviour is part of string interning, which helps conserve memory by reusing identical string instances.

Immutable String in Java

In Java, string objects are immutable, which means they cannot be modified or changed after creation. Instead, any operations that appear to modify a string actually create a new string object with the desired changes.

```
class Demo {
   public static void main(String[] args)
   {
      String name = "Sachin";
      name.concat(" Tendulkar");
      System.out.println(name);
   }
}
```

Output :-

```
Sachin
```

```
class Demo {
   public static void main(String[] args)
   {
      String name = "Sachin";
       name = name.concat(" Tendulkar");
      System.out.println(name);
   }
}
```

Output :-

```
Sachin Tendulkar
```

JAVA Input and Output streams(I/O)

Java I/O, which stands for Input and Output, is a fundamental part of Java programming used for handling data input and producing output in various applications.

Java leverages the concept of streams to optimise I/O operations, and the java.io package encompasses all the necessary classes for handling input and output tasks efficiently.

File handling in Java can be achieved through the Java I/O API.

Stream

In Java, a stream is a sequence of elements that you can process in a functional and declarative manner. Java Streams were introduced in Java 8 and provide a powerful way to work with collections, arrays, and other data sources. Streams allow you to perform operations on the elements of a sequence, such as filtering, mapping, and reducing, without the need for explicit loops.
In Java, three streams are automatically created for us, all of which are associated with the console:

1) `System.in`: This represents the standard input stream, which serves the purpose of reading characters from the keyboard or any other standard input source.

2)`System.out`: This refers to the standard output stream, utilised for displaying the program's results on an output device, such as the computer screen. Here is a list of various print functions used for outputting statements

- **'print()':** In Java, this method is employed to exhibit text on the console. The text is supplied as a parameter in the form of a string. When invoked, this method displays the text on the console while keeping the cursor at the end of the printed text. Subsequent printing operations will begin from this point.

Syntax:
System.out.print(parameter);

```
// Java code to illustrate print()
import java.io.*;

class Demo {
   public static void main(String[] args)
   {
        // using print()
        // all are printed in the
        // same line
        System.out.print("CTI WORLD ");
        System.out.print("CTI WORLD ");
        System.out.print("CTI WORLD ");
    }
}
```

Output:

```
CTI WORLD CTI WORLD CTI WORLD
```

- **println():** In Java, this method serves the purpose of showcasing text on the console. It outputs the text to the console and positions the cursor at the beginning of the next line. Subsequent printing operations will begin from the next line

Syntex:-

System.out.println(parameter);

```
// Java code to illustrate println()

import java.io.*;
class Demo {
    public static void main(String[] args)
    {

        // using println()
        // all are printed in the
        // different line
        System.out.println("CSA WORLD ");
        System.out.println("CSA WORLD ");
        System.out.println("CSA WORLD ");
    }
}
```

Output:

```
CSA WORLD
CSA WORLD
CSA WORLD
```

- **printf():** This method in Java is reminiscent of the printf function in C. It is worth noting that while System.out.print() and System.out.println() accept a single argument, printf() can accept multiple arguments. Its primary purpose is to format the output in Java.

```
public class PrintfExample {
            public static void main(String[] args) {
      String name = "ANSHU";
      int age = 24;
      double salary = 50000.75;

      // Using printf to format output
    System.out.printf("Name: %s%n", name);
    System.out.printf("Age: %d%n", age);
    System.out.printf("Salary: $%.2f%n", salary);
     }
 }
```

Output:-

```
Name: ANSHU
Age: 34
tSalary: 50000.75
```

In this example:

- **%s** is a placeholder for a string (name).
- **%d** is a placeholder for an integer (age).
- **%.2f** is a placeholder for a floating-point number (salary) with 2 decimal places.

The %n is used to insert a platform-independent line separator.

Types of Streams :-

Streams can be categorised into two main classes based on the type of operations they are used for

1.Input Stream: These streams are employed to retrieve data as input from various sources such as arrays, files, or peripheral devices. Examples include FileInputStream, BufferedInputStream, and ByteArrayInputStream

2.Output Stream: These streams are used to write data as output to various destinations such as arrays, files, or output peripheral devices. Examples include FileOutputStream, BufferedOutputStream, and ByteArrayOutputStream

Input using Scanner class and Console class methods

In Java, you can use the Scanner class and the Console class to interact with the user via the command line for input and output. Here are examples of how to use both classes for input:

Scanner class

In Java, the Scanner class from the java.util package is employed for acquiring input of primitive types such as int, double, as well as strings.

While using the Scanner class is the simplest method for reading input in a Java program, it may not be the most efficient choice for situations where input processing time is critical, such as competitive programming.

Syntax:-

Scanner sc=new Scanner(System.in);

Methods of Java Scanner Class

- nextBoolean(): Used for reading a Boolean value.
- nextByte(): Used for reading a Byte value.
- nextDouble(): Used for reading a Double value.
- nextFloat(): Used for reading a Float value.
- nextInt(): Used for reading an Int value.
- nextLine(): Used for reading a Line value (usually a String).
- nextLong(): Used for reading a Long value.
- nextShort(): Used for reading a Short value.

"Let's examine a code snippet that demonstrates how to read input of various data types."

Example:-

// Java program to read data of various types using Scanner

```
// class.
import java.util.Scanner;
public class ScannerDemo1 {
    public static void main(String[] args)
    {
        // Declare the object and initialise with
        // predefined standard input object
        Scanner sc = new Scanner(System.in);

        // String input
        String name = sc.nextLine();

        // Character input
        char gender = sc.next().charAt(0);

        // Numerical data input
        // byte, short and float can be read
        // using similar-named functions.
        int age = sc.nextInt();
        double cgpa = sc.nextDouble();

        // Print the values to check if the input was
        // correctly obtained.
        System.out.println("Name: " + name);
        System.out.println("Gender: " + gender);
        System.out.println("Age: " + age);
        System.out.println("CGPA: " + cgpa);
    }
}
```

1.It imports the Scanner class to facilitate user input.

2.In the main method:

- It creates a Scanner object sc to read input from the standard input stream (System.in).
- It uses sc.nextLine() to read a line of input as a String and stores it in the name variable.

- It uses sc.next().charAt(0) to read the next token (word) as a String and then extracts the first character of that string as a char, storing it in the gender variable.
- It uses sc.nextInt() to read the next token as an int and stores it in the age variable.
- It uses sc.nextDouble() to read the next token as a double and stores it in the cgpa variable.

3.After reading all the values, it prints them to the console to verify that the input was correctly obtained.

Input

```
AKASH MOHAN
Male
24
91
```

Output

```
Name: AKASH MOHAN
Gender: M
Age: 24
CGPA: 91.0
```

Sometimes, it's necessary to verify if the next value we're about to read is of a specific data type or if the input has reached its end (EOF marker encountered).

To accomplish this, we can utilise the hasNextXYZ() functions, where XYZ represents the type we are interested in. These functions return true if the Scanner has a token of the specified type, and false otherwise. For example, in the code below, we have employed hasNextInt() to check for an integer input. To check for a string, we use hasNextLine(), and for a single character, we use hasNext().charAt(0)
Let's review a code snippet for reading numbers from the console and calculating their mean

```
// Java program to read some values using Scanner
// class and print their mean.
import java.util.Scanner;

public class ScannerDemo2 {
    public static void main(String[] args)
    {
        // Declare an object and initialise with
        // predefined standard input object
        Scanner sc = new Scanner(System.in);

        // Initialize sum and count of input elements
        int sum = 0, count = 0;

        // Check if an int value is available
        while (sc.hasNextInt()) {
            // Read an int value
            int num = sc.nextInt();
            sum += num;
            count++;
        }
        if (count > 0) {
            int mean = sum / count;
            System.out.println("Mean: " + mean);
        }
        else {
            System.out.println(
                "No integers were input. Mean cannot be calculated.");
        }
    }
}
```

This Java program reads integer values from the standard input using the Scanner class and calculates their mean (average). Here's a breakdown of how the program works:

1.It imports the Scanner class to facilitate user input.

2.In the main method:

- It creates a Scanner object sc to read input from the standard input stream (System.in).
- Initialises two variables, sum and count, to keep track of the sum of input values and the count of input values, respectively.

- Uses a while loop to continuously check if the next input is an integer using sc.hasNextInt(). If an integer is available, it reads the integer value using sc.nextInt(), adds it to the sum, and increments the count.

3.After the loop, it checks whether any integers were input (i.e., count > 0). If count is greater than zero, it calculates the mean (average) by dividing sum by count and then prints the mean value. If no integers were input (i.e., count is still zero), it prints a message stating that the mean cannot be calculated.

Input:

```
1

2

3

4

5
```

Output:

```
Mean: 3
```

- Importing the Class: To use the Scanner class, you need to import it with import java.util.Scanner;.
- Reading from Standard Input: The Scanner class is commonly used to read input from the standard input stream (System.in) by creating a Scanner object with System.in as the argument.
- Reading from Files: You can also read input from files by passing a File object as an argument to the Scanner constructor.
- Reading Different Data Types: Scanner provides methods like nextByte(), nextInt(), nextLong(), nextFloat(), nextDouble(), etc., to read different data types.
- Reading Strings: To read strings, you can use next() or nextLine() methods. next() reads the next token (usually a word), while nextLine() reads the entire line.
- Reading Characters: You can read a single character by combining next() and charAt(0), as in next().charAt(0).
- Delimiter: By default, Scanner uses whitespace as the delimiter to separate tokens. You can change the delimiter using the useDelimiter() method.

- Checking for Input: You can check if there is more input to read with methods like hasNext(), hasNextInt(), hasNextLine(), etc.
- Closing the Scanner: It's important to close the Scanner object using close() when you're done with it to release resources.
- Handling Input Errors: Always validate input using methods like hasNextInt() or exception handling to prevent unexpected program behaviour due to invalid input.
- Tokenization: Scanner reads input and tokenizes it. Tokens are smaller units of input separated by the specified delimiter.
- Locale and Locale-Sensitive Parsing: You can set the locale to influence how Scanner parses numbers and strings. For example, comma (,) or period (.) as a decimal separator.
- Not Suitable for Competitive Programming: While Scanner is convenient, it may not be the most efficient choice for scenarios where input processing time is critical, such as competitive programming.
- Exception Handling: Be prepared to handle exceptions like InputMismatchException or NoSuchElementException that may occur if the input doesn't match the expected format.
- Thread Safety: Scanner is not thread-safe, so avoid sharing Scanner objects between multiple threads without proper synchronisation.
- Unicode Support: Scanner supports Unicode characters, allowing you to read and process text in various languages.

JAVA Program Flow Control

• Decision making and flow control using if…then, if then else, nested if, switch case and the conditional ternary operators in JAVA.

• Loop control flow using while – do, do – while loops, for loop, using the break, continue statements.

• Terminating the JAVA program.

• JAVA Number, Character and String Classes.

• Arrays in JAVA.

Java Control Statements | Control Flow in Java

Java compiler executes the code from top to bottom. The statements in the code are executed according to the order in which they appear. However, Java provides statements that can be used to control the flow of Java code. Such statements are called control flow statements. It is one of the fundamental features of Java, which provides a smooth flow of program.

Java provides three types of control flow statements.

1. Decision Making statements
    - if statements
    - switch statement
2. Loop statements
    - do while loop
    - while loop
    - for loop
    - for-each loop
3. Jump statements
    - break statement
    - continue statement

Decision-Making statements:

As the name suggests, decision-making statements decide which statement to execute and when. Decision-making statements evaluate the Boolean expression and control the program flow depending upon the result of the condition provided. There are two types of decision-making statements in Java, i.e., If statement and switch statement.

1) If Statement:

In Java, the "if" statement is used to evaluate a condition. The control of the program is diverted depending upon the specific condition. The condition of the If statement gives a Boolean value, either true or false. In Java, there are four types of if-statements given below.

1. Simple if statement
2. if-else statement

3. if-else-if ladder
4. Nested if-statement

Let's understand the if-statements one by one.

1) Simple if statement:

It is the most basic statement among all control flow statements in Java. It evaluates a Boolean expression and enables the program to enter a block of code if the expression evaluates to true. Syntax of if statement is given below.

```
if(condition) {
statement 1; //executes when condition is true
}
```

Consider the following example in which we have used the **if** statement in the java code.
Student.java

Student.java

```
public class Student {
public static void main(String[] args) {
int x = 10;
int y = 12;
if(x+y > 20) {
System.out.println("x + y is greater than 20");
}
}
}
```

Output:

```
x + y is greater than 20
```

2) if-else statement

The if-else statement is an extension to the if-statement, which uses another block of code, i.e., else block. The else block is executed if the condition of the if-block is evaluated as false.

Syntax:

```
if(condition) {
statement 1; //executes when condition is true
}
else{
statement 2; //executes when condition is false
}
```

Consider the following example.

Student.java

```
public class Student {
public static void main(String[] args) {
int x = 10;
int y = 12;
if(x+y < 10) {
System.out.println("x + y is less than      10");
}   else {
System.out.println("x + y is greater than 20");
}
}
}
```

Output:

```
x + y is greater than 20
```

3) if-else-if ladder:

The if-else-if statement contains the if-statement followed by multiple else-if statements. In other words, we can say that it is the chain of if-else statements that create a decision tree where the program may enter in the block of code where the condition is true. We can also define an else statement at the end of the chain.

Syntax of if-else-if statement is given below.

```
if(condition 1) {
statement 1; //executes when condition 1 is true
}
```

```
else if(condition 2) {
statement 2; //executes when condition 2 is true
}
else {
statement 2; //executes when all the conditions are false
}
```

Consider the following example.

Student.java

```
public class Student {
public static void main(String[] args) {
String city = "Delhi";
if(city == "Meerut") {
System.out.println("city is meerut");
}else if (city == "Noida") {
System.out.println("city is noida");
}else if(city == "Agra") {
System.out.println("city is agra");
}else {
System.out.println(city);
}
}
}
```

Output:

```
Delhi
```

. Nested if-statement

In nested if-statements, the if statement can contain a **if** or **if-else** statement inside another if or else-if statement.

Syntax of Nested if-statement is given below.

```
if(condition 1) {
statement 1; //executes when condition 1 is true
if(condition 2) {
```

```
statement 2; //executes when condition 2 is true
}
else{
statement 2; //executes when condition 2 is false
}
}
```

Consider the following example.

Student.java

```
public class Student {
public static void main(String[] args) {
String address = "Delhi, India";

if(address.endsWith("India")) {
if(address.contains("Meerut")) {
System.out.println("Your city is Meerut");
}else if(address.contains("Noida")) {
System.out.println("Your city is Noida");
}else {
System.out.println(address.split(",")[0]);
}
}else {
System.out.println("You are not living in India");
}
}
}
```

Output:

```
Delhi
```

Switch Statement:

In Java, Switch statements are similar to if-else-if statements. The switch statement contains multiple blocks of code called cases and a single case is executed based on the variable which is being switched. The switch statement is easier to use instead of if-else-if statements. It also enhances the readability of the program.

Points to be noted about switch statement:

- The case variables can be int, short, byte, char, or enumeration. String type is also supported since version 7 of Java
- Cases cannot be duplicate
- Default statement is executed when any of the case doesn't match the value of expression. It is optional.
- Break statement terminates the switch block when the condition is satisfied.
  It is optional, if not used, next case is executed.
- While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value.

The syntax to use the switch statement is given below.

```
switch (expression){
    case value1:
     statement1;
     break;
    .
    .
    .
    case valueN:
     statementN;
     break;
    default:
     default statement;
}
```

Consider the following example to understand the flow of the switch statement.

Student.java

```
public class Student implements Cloneable {
public static void main(String[] args) {
```

```
int num = 2;
switch (num){
case 0:
System.out.println("number is 0");
break;
case 1:
System.out.println("number is 1");
break;
default:
System.out.println(num);
}
}
}
```

Output:

```
2
```

While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value. The switch permits only int, string, and Enum type variables to be used.

Loop Statements

In programming, sometimes we need to execute the block of code repeatedly while some condition evaluates to true. However, loop statements are used to execute the set of instructions in a repeated order. The execution of the set of instructions depends upon a particular condition.

In Java, we have three types of loops that execute similarly. However, there are differences in their syntax and condition checking time.

1. for loop
2. while loop
3. do-while loop

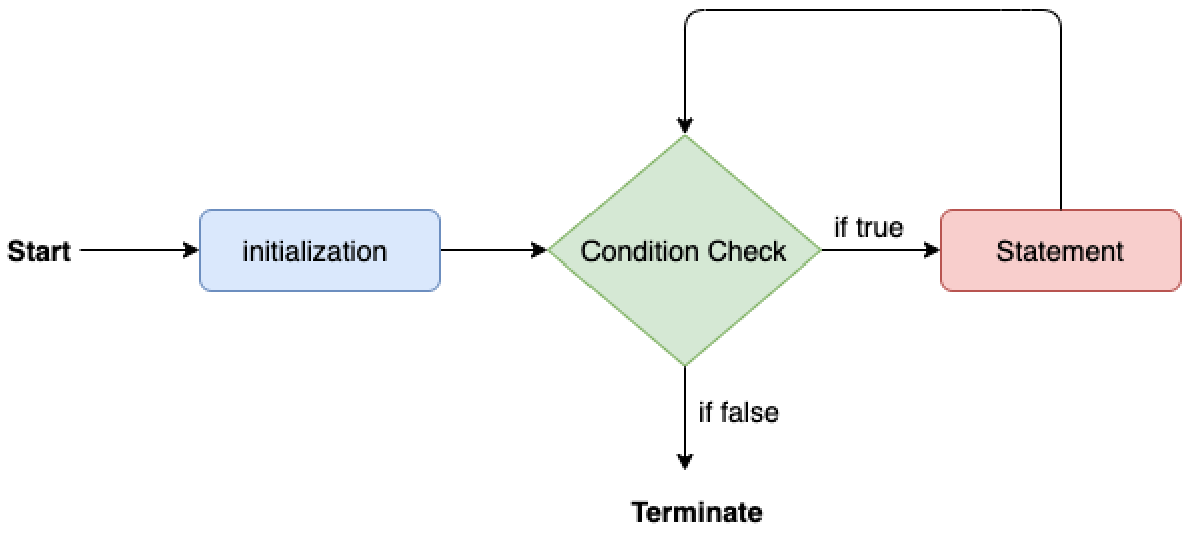
Let's understand the loop statements one by one.

Java for loop

In Java, for loop is similar to C and C++. It enables us to initialize the loop variable, check the condition, and increment/decrement in a single line of code. We use the for loop only when we exactly know the number of times, we want to execute the block of code.

```
for(initialization, condition, increment/decrement) {
//block of statements
}
```

The flow chart for the for-loop is given below.



Consider the following example to understand the proper functioning of the for loop in java.

Calculation.java

ADVERTISEMENT

ADVERTISEMENT

```
public class Calculattion {
public static void main(String[] args) {
// TODO Auto-generated method stub
int sum = 0;
for(int j = 1; j<=10; j++) {
sum = sum + j;
}
System.out.println("The sum of first 10 natural numbers is " + sum);
}
}
```

Output:

```
The sum of first 10 natural numbers is 55
```

Java for-each loop

Java provides an enhanced for loop to traverse the data structures like array or collection. In the for-each loop, we don't need to update the loop variable. The syntax to use the for-each loop in java is given below.

```
for(data_type var : array_name/collection_name){
//statements
}
```

Consider the following example to understand the functioning of the for-each loop in Java.

Calculation.java

```
public class Calculation {
public static void main(String[] args) {
// TODO Auto-generated method stub
String[] names = {"Java","C","C++","Python","JavaScript"};
System.out.println("Printing the content of the array names:\n");
for(String name:names) {
System.out.println(name);
}
}
}
```

Output:

```
Printing the content of the array names:
```

Java

C

C++

Python

JavaScript

Java while loop

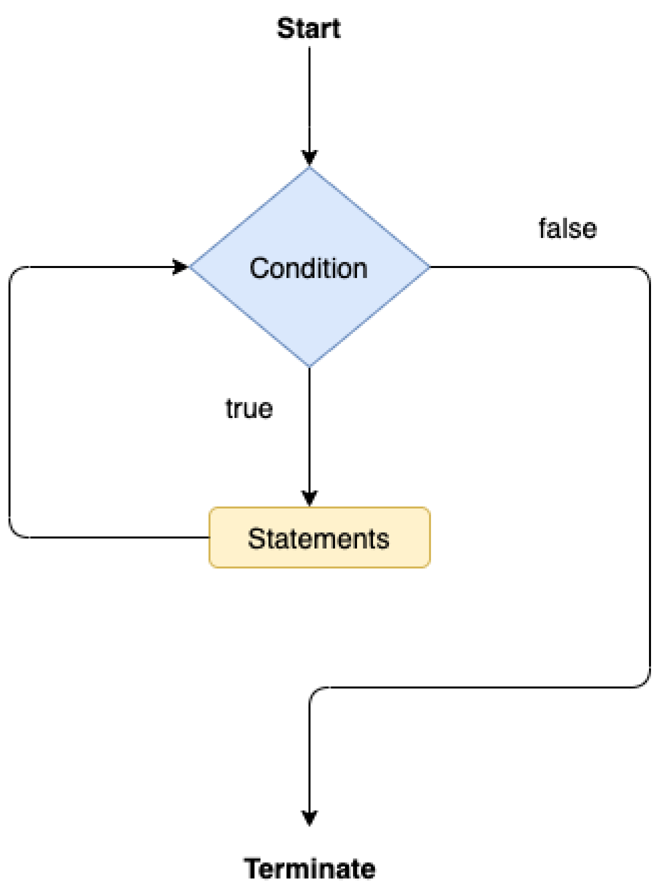
The while loop is also used to iterate over the number of statements multiple times. However, if we don't know the number of iterations in advance, it is recommended to use a while loop. Unlike for loop, the initialization and increment/decrement doesn't take place inside the loop statement in while loop.

It is also known as the entry-controlled loop since the condition is checked at the start of the loop. If the condition is true, then the loop body will be executed; otherwise, the statements after the loop will be executed.

The syntax of the while loop is given below.

```
while(condition){
//looping statements
}
```

The flow chart for the while loop is given in the following image.



Consider the following example.

Calculation .java

```java
public class Calculation {
public static void main(String[] args) {
// TODO Auto-generated method stub
int i = 0;
System.out.println("Printing the list of first 10 even numbers \n");
while(i<=10) {
System.out.println(i);
i = i + 2;
}
}
}
```

Output:

```
Printing the list of first 10 even numbers
```

```
0

2

4

6

8

10
```

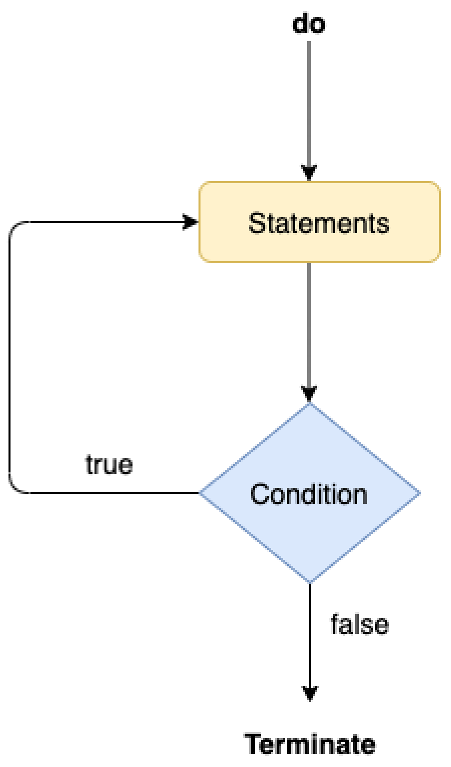
Java do-while loop

The do-while loop checks the condition at the end of the loop after executing the loop statements. When the number of iteration is not known and we have to execute the loop at least once, we can use do-while loop.

It is also known as the exit-controlled loop since the condition is not checked in advance. The syntax of the do-while loop is given below.

```
do
{
//statements
} while (condition);
```

The flow chart of the do-while loop is given in the following image.



Consider the following example to understand the functioning of the do-while loop in Java.

Calculation.java

```
public class Calculation {
public static void main(String[] args) {
// TODO Auto-generated method stub
int i = 0;
System.out.println("Printing the list of first 10 even numbers \n");
do {
System.out.println(i);
i = i + 2;
}while(i<=10);
}
}
```

Output:

```
Printing the list of first 10 even numbers

0
```

```
2

4

6

8

10
```

Jump Statements

Jump statements are used to transfer the control of the program to the specific statements. In other words, jump statements transfer the execution control to the other part of the program. There are two types of jump statements in Java, i.e., break and continue.

Java break statement

As the name suggests, the break statement is used to break the current flow of the program and transfer the control to the next statement outside a loop or switch statement. However, it breaks only the inner loop in the case of the nested loop.

The break statement cannot be used independently in the Java program, i.e., it can only be written inside the loop or switch statement.

The break statement example with for loop

Consider the following example in which we have used the break statement with the for loop.

BreakExample.java

```java
public class BreakExample {

public static void main(String[] args) {
// TODO Auto-generated method stub
for(int i = 0; i<= 10; i++) {
System.out.println(i);
if(i==6) {
break;
}
}
}
}
```

Output:

```
0

1

2

3

4

5

6
```

break statement example with labeled for loop

Calculation.java

```java
public class Calculation {

public static void main(String[] args) {
// TODO Auto-generated method stub
a:
for(int i = 0; i<= 10; i++) {
b:
for(int j = 0; j<=15;j++) {
c:
for (int k = 0; k<=20; k++) {
System.out.println(k);
if(k==5) {
break a;
}
}
}

}
}


}
```

Output:

```
0

1

2

3

4

5
```

Java continue statement

Unlike break statement, the continue statement doesn't break the loop, whereas, it skips the specific part of the loop and jumps to the next iteration of the loop immediately.

Consider the following example to understand the functioning of the continue statement in Java.

```
public class ContinueExample {

public static void main(String[] args) {
// TODO Auto-generated method stub

for(int i = 0; i<= 2; i++) {

for (int j = i; j<=5; j++) {
```

```

if(j == 4) {
continue;
}
System.out.println(j);
}
}
}

}
```

Output:

```
0
1
2
3
5
1
2
3
5
2
3
5
```

Terminate a Java program

What is exit() Method in Java?

Java provides exit() method to exit the program at any point by terminating running JVM, based on some condition or programming logic. In Java exit() method is in java.lang.System class. This System.exit() method terminates the current JVM running on the system which results in termination of code being executed currently. This method takes status code as an argument.

Uses

- exit() method is required when there is an abnormal condition and we need to terminate the program immediately.
- exit() method can be used instead of throwing an exception and providing the user a script that mentions what caused the program to exit abruptly.
- Another use-case for exit() method in java, is if a programmer wants to terminate the execution of the program in case of wrong inputs.

Syntax of exit() in Java

public static void exit(int status)

Parameters of exit() in jJava

System.exit() method takes status as a parameter, these status codes tell about the status of termination. This status code is an integer value and its meanings are as follows:

- exit(0) - Indicates successful termination
- exit(1) - Indicates unsuccessful termination
- exit(-1) - Indicates unsuccessful termination with Exception

Note: Any non-zero value as status code indicates unsuccessful termination.

Return Value for exit() in Java

As the syntax suggests, it is a void method that does not return any value.

Exit a Java Method using System.exit(0)

As mentioned earlier, System.exit(0) method terminates JVM which results in termination of the currently running program too. Status is the single parameter that the method takes. If the status is 0, it indicates the termination is successful.
Let's see practical implementations of System.exit(0) method in Java.
Example 1
Here is a simple program that uses System.exit(0) method to terminate the program based on the condition.

```
public class SampleExitMethod {

  public static void exampleMethod(int[] array1) {
    for (int i = 0; i < array1.length; i++) {
      //Check condition for i
      if (i > 4) {
        System.out.println("Terminating JVM...");

        //Terminates JVM when if condition is satisfied
        System.exit(0);
      }
      System.out.println("Array Index: " + i + " Array Element: " + array1[i]);
    }
  }

  public static void main(String[] args) {
    int[] array1 = { 0, 2, 4, 6, 8, 10, 12, 14, 16 };

    //function call
    exampleMethod(array1);
  }
}
```

Output:

```
Array Index: 0 Array Element: 0
Array Index: 1 Array Element: 2
Array Index: 2 Array Element: 4
Array Index: 3 Array Element: 6
Array Index: 4 Array Element: 8
Terminating JVM...
```

Explanation:

In the above program, an array is being accessed till its index is 4. Until if condition is false i.e., array index <= 4, JVM will give Array index and element stored at that index as the output.

Once the if condition is true i.e., array index > 4, JVM will execute statements inside the if condition. Here it will first execute the print statement and when exit(0) method is called it will terminate JVM and program execution.

Example 2

Let's see one more example of exit() method in a try-catch-finally block.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class check {

  public static void main(String[] args) {
    try {
      //Reads "file.txt" file
      BufferedReader br = new BufferedReader(new FileReader("file.txt"));

      // Closes bufferreader br
      System.out.println(br.readLine());
      br.close();

    } // Catches exception
    catch (IOException e) {
      System.out.println("Exception caught. Terminating JVM.");
      System.exit(0);
    } // Terminates JVM as file is not available
    finally {
      System.out.println("Exiting the program");
    }
  }
}
```

Output:

```
Exception caught. Terminating JVM.
```

Explanation:

The piece of code above is trying to read a file and print a line from it if the file exists. If a file doesn't exist, the exception is raised and execution will go in catch block and code exits with System.exit(0) method in the catch block.

- We know, finally block is executed every time irrespective of try-catch blocks are being executed or not. But, in this case, the file is not present so JVM will check the catch block where System.exit(0) method is used that terminates JVM and currently running the program.
- As the JVM is terminated in the catch block, JVM won't be able to read the final block hence, the final block will not be executed.

Exit a Java Method using Return

exit() method in java is the simplest way to terminate the program in abnormal conditions. There is one more way to exit the java program using return keyword. return keyword completes execution of the method when used and returns the value from the function. The return keyword can be used to exit any method when it doesn't return any value.

Let's see this using an example:

```
public class SampleExitMethod2 {

  public static void SubMethod(int num1, int num2) {
    if (num2 > num1) return;

    int answer = num1 - num2;
    System.out.println(answer);
  }

  public static void main(String[] args) {
    SubMethod(2, 5); // if condition is true
    SubMethod(3, 2);
    SubMethod(100, 20);
    SubMethod(102, 110); // if condition is true
  }
}
```

Output:

```
1
80
```

Explanation:

In the above program, SubMethod takes two arguments num1 and num2, subtracts num2 from num1 giving the result in the answer variable. But it has one condition that states num2 should not be greater than num1, this condition is implemented using if conditional block. If this condition is true, it will execute if block which has a return statement. In this case, SubMethod is of void type hence, return doesn't return any value and in this way exits the function.

Does goto Exist in Java?

No, goto does not exist in Java. Because Java has reserved it for now and may use it in a later version.

In other languages, goto is a control statement used to transfer control to a certain point in the programming. After goto statement is executed program starts executing the lines where goto has transferred its control and then other lines are executed. goto statement works with labels that are identifiers, these labels are used to the state where goto has to transfer the control of execution.

Java does not support goto() method but it supports label. No support for goto method is due to the following reasons:

1. Goto-ridden code hard to understand and hard to maintain
2. These labels can be used with nested loops. A combination of break, continue and labels can be used to achieve the functionality of goto() method in Java.
3. Prohibits compiler optimization

Example:

```
// Java code
```

```
public class Main {

  public static void main(String[] args) {
    boolean t = true;
    first:{
      second:{
        third:{
          System.out.println("Before the break statement");
          if (
            t
          ) break second; // break out of second block
        }
        System.out.println("No execution for this statement");
      }
      System.out.println("Part of first block, outside second block.");
    }
  }
}
```

Output:

```
Before the break statement
Part of first block, outside second block.
```

In the above code, the outer label is created for the first for loop. When if the condition is true break outer; statement will go back to first for loop and break the execution. Use this Compiler to compile your Java code.

Conclusion

- exit() method is used to terminate JVM.
- Status code other than 0 in exit() method indicates abnormal termination of code.
- goto does not exist in Java, but it supports labels.
- It is better to use exception handling or plain return statements to exit a program while execution.

- System.exit() method suit better for script-based applications or wherever the status codes are interpreted.

Java Numbers

Numbers

Primitive number types are divided into two groups:
Integer types stores whole numbers, positive or negative (such as 123 or -456), without decimals. Valid types are byte, short, int and long. Which type you should use, depends on the numeric value.
Floating point types represents numbers with a fractional part, containing one or more decimals. There are two types: float and double

Integer Types

Byte

The byte data type can store whole numbers from -128 to 127. This can be used instead of int or other integer types to save memory when you are certain that the value will be within -128 and 127:

Example

```
byte myNum = 100;
System.out.println(myNum);
```

Short

The short data type can store whole numbers from -32768 to 32767:

Example

```
short myNum = 5000;
System.out.println(myNum);
```

Int

The int data type can store whole numbers from -2147483648 to 2147483647. In general, and in our tutorial, the int data type is the preferred data type when we create variables with a numeric value.

Example

```
int myNum = 100000;
System.out.println(myNum);
```

Long

The long data type can store whole numbers from -9223372036854775808 to 9223372036854775807. This is used when int is not large enough to store the value. Note that you should end the value with an "L":

Example

```
long myNum = 15000000000L;
System.out.println(myNum);
```

Floating Point Types

You should use a floating point type whenever you need a number with a decimal, such as 9.99 or 3.14515.
The float and double data types can store fractional numbers. Note that you should end the value with an "f" for floats and "d" for doubles:

Example

```
float myNum = 5.75f;
System.out.println(myNum);
```

Double Example

Example

double myNum = 19.99d;

System.out.println(myNum);

Java String

In Java, string is basically an object that represents sequence of char values. An array of characters works same as Java string

For example:

1. char[] ch={'j','a','v','a','t','p','o','i','n','t'};
2. String s=new String(ch);

Java String class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

The java.lang.String class implements Serializable, Comparable and CharSequence interfaces.

What is String in Java?

Generally, String is a sequence of characters. But in Java, string is an object that represents a sequence of characters. The java.lang.String class is used to create a string object.

How to create a string object?

There are two ways to create String object:

1. By string literal
2. By new keyword

1) String Literal

Java String literal is created by using double quotes.

For example:

1. String s="welcome";

Why Java uses the concept of String literal?

To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

2) By new keyword

```java
String s=new String("Welcome");//creates two objects and one reference variable
```

In such case, JVM will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).

Java String Example

StringExample.java

```java
public class StringExample{
public static void main(String args[]){
String s1="java";//creating string by Java string literal
char ch[]={'s','t','r','i','n','g','s'};
String s2=new String(ch);//converting char array to string
String s3=new String("example");//creating Java string by new keyword
System.out.println(s1);
System.out.println(s2);
System.out.println(s3);
}}
```

Output:

```
java
strings
example
```

Java Arrays

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

To declare an array, define the variable type with square brackets:

String[] cars;

We have now declared a variable that holds an array of strings. To insert values to it, you can place the values in a comma-separated list, inside curly braces:

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

To create an array of integers, you could write:

```
int[] myNum = {10, 20, 30, 40};
```

Access the Elements of an Array

You can access an array element by referring to the index number.
This statement accesses the value of the first element in cars:

Example

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
System.out.println(cars[0]);
// Outputs Volvo
```

Change an Array Element

To change the value of a specific element, refer to the index number:

Example

```
cars[0] = "Opel";
```

Java Classes, Overloading and Inheritance

Java - What is OOP?

OOP stands for Object-Oriented Programming.
Procedural programming is about writing procedures or methods that perform operations on the data, while object-oriented programming is about creating objects that contain both data and methods.
Object-oriented programming has several advantages over procedural programming:

- OOP is faster and easier to execute
- OOP provides a clear structure for the programs
- OOP helps to keep the Java code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter development time

JAVA Classes, Overloading and Inheritance

- JAVA Objects, Classes and Methods.
- Passing data and objects as parameters to methods.
- Method Overloading.
- Constructors and Overloaded constructors.
- Inheritance in JAVA.
- Method Overriding in JAVA.

Java Classes/Objects

Classes and objects are the two main aspects of object-oriented programming.
Java is an object-oriented programming language.
Everything in Java is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an object. The car has attributes, such as weight and color, and methods, such as drive and brake.
A Class is like an object constructor, or a "blueprint" for creating objects

Create a Class

To create a class, use the keyword class:

Main.java

Create a class named "Main" with a variable x:

```
public class Main {
  int x = 5;
}
```

Create an Object

In Java, an object is created from a class. We have already created the class named Main, so now we can use this to create objects.
To create an object of Main, specify the class name, followed by the object name, and use the keyword new:

Example

Create an object called "myObj" and print the value of x:

```
public class Main {
  int x = 5;

  public static void main(String[] args) {
    Main myObj = new Main();
    System.out.println(myObj.x);
  }
}
```

Java Methods

A method is a block of code which only runs when it is called.
You can pass data, known as parameters, into a method.
Methods are used to perform certain actions, and they are also known as functions.
Why use methods? To reuse code: define the code once, and use it many times.

Create a Method

A method must be declared within a class. It is defined with the name of the method, followed by parentheses (). Java provides some pre-defined methods, such as System.out.println(), but you can also create your own methods to perform certain actions:

Example

Create a method inside Main:

```
public class Main {
  static void myMethod() {
    // code to be executed
  }
}
```

Example Explained

- myMethod() is the name of the method
- static means that the method belongs to the Main class and not an object of the Main class. You will learn more about objects and how to access methods through objects later in this tutorial.
- void means that this method does not have a return value

Call a Method

To call a method in Java, write the method's name followed by two parentheses () and a semicolon;
In the following example, myMethod() is used to print a text (the action), when it is called:

Example

```
public class Main {
  static void myMethod() {
    System.out.println("I just got executed!");
  }

  public static void main(String[] args) {
    myMethod();
  }
}

// Outputs "I just got executed!"
```

Java Object as Parameter

Objects, like primitive types, can be passed as parameters to methods in Java. When passing an object as a parameter to a method, a reference to the object is passed rather than a copy of the object itself. This means that any modifications made to the object within the method will have an impact on the original object.

```
public class MyClass {
    // Fields or attributes
    private int attribute1;
    private String attribute2;
    private double attribute3;

    // Constructor
    public MyClass(int attribute1, String attribute2, double attribute3) {
        this.attribute1 = attribute1;
        this.attribute2 = attribute2;
        this.attribute3 = attribute3;
    }

    // Method with object as parameter
    public void myMethod(MyClass obj) {
        // block of code to define this method
    }

    // More methods
}
```

Java Method Overloading

In Java, two or more methods may have the same name if they differ in parameters (different number of parameters, different types of parameters, or both). These methods are called overloaded methods and this feature is called method overloading.

Why method overloading?

Suppose, you have to perform the addition of given numbers but there can be any number of arguments (let’s say either 2 or 3 arguments for simplicity).
In order to accomplish the task, you can create two methods sum2num(int, int) and sum3num(int, int, int) for two and three parameters respectively. However, other programmers, as well as you in the future may get confused as the behavior of both methods are the same but they differ by name.
The better way to accomplish this task is by overloading methods. And, depending upon the argument passed, one of the overloaded methods is called. This helps to increase the readability of the program

How to perform method overloading in Java?

Here are different ways to perform method overloading:

```
class MethodOverloading {
    private static void display(int a){
        System.out.println("Arguments: " + a);
    }

    private static void display(int a, int b){
        System.out.println("Arguments: " + a + " and " + b);
    }

    public static void main(String[] args) {
        display(1);
        display(1, 4);
    }
}
```

```
Output:
Arguments: 1
Arguments: 1 and 4
```

Important Points

- Two or more methods can have the same name inside the same class if they accept different arguments. This feature is known as method overloading.
- Method overloading is achieved by either:
    - changing the number of arguments.
    - or changing the data type of arguments.
- It is not method overloading if we only change the return type of methods. There must be differences in the number of parameters.

Java Constructors

A constructor in Java is similar to a method that is invoked when an object of the class is created.

Unlike Java methods, a constructor has the same name as that of the class and does not have any return type. For example,

```
class Test {
  Test() {
    // constructor body
  }
}
```

Example: Java Constructor

```
class Main {
  private String name;

  // constructor
  Main() {
    System.out.println("Constructor Called:");
    name = "Programiz";
  }

  public static void main(String[] args) {

    // constructor is invoked while
    // creating an object of the Main class
    Main obj = new Main();
    System.out.println("The name is " + obj.name);
  }
}
```

Output:

```
Constructor Called:
The name is Programiz
```

Types of Constructor

In Java, constructors can be divided into three types:

1. No-Arg Constructor
2. Parameterized Constructor
3. Default Constructor

1. Java No-Arg Constructors

Similar to methods, a Java constructor may or may not have any parameters (arguments).

If a constructor does not accept any parameters, it is known as a no-argument constructor.

2. Java Parameterized Constructor

A Java constructor can also accept one or more parameters. Such constructors are known as parameterized constructors (constructors with parameters).

3. Java Default Constructor

If we do not create any constructor, the Java compiler automatically creates a no-arg constructor during the execution of the program.

Important Notes on Java Constructors

- Constructors are invoked implicitly when you instantiate objects.
- The two rules for creating a constructor are:
  1. The name of the constructor should be the same as the class.
  2. A Java constructor must not have a return type.
- If a class doesn't have a constructor, the Java compiler automatically creates a default constructor during run-time. The default constructor initializes instance variables with default values. For example, the int variable will be initialized to 0
- Constructor types:
  No-Arg Constructor - a constructor that does not accept any arguments
  Parameterized constructor - a constructor that accepts arguments
  Default Constructor - a constructor that is automatically created by the Java compiler if it is not explicitly defined.
- A constructor cannot be abstract or static or final.
- A constructor can be overloaded but can not be overridden.

Constructors Overloading in Java

Similar to Java method overloading, we can also create two or more constructors with different parameters. This is called constructor overloading.

Example: Java Constructor Overloading

```
class Main {

  String language;

  // constructor with no parameter
  Main() {
    this.language = "Java";
  }

  // constructor with a single parameter
  Main(String language) {
    this.language = language;
  }

  public void getName() {
    System.out.println("Programming Language: " + this.language);
  }

  public static void main(String[] args) {

    // call constructor with no parameter
    Main obj1 = new Main();

    // call constructor with a single parameter
    Main obj2 = new Main("Python");

    obj1.getName();
    obj2.getName();
  }
}
```

Output

```
Programming Language: Java
```

Programming Language: Python

Inheritance in Java

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.

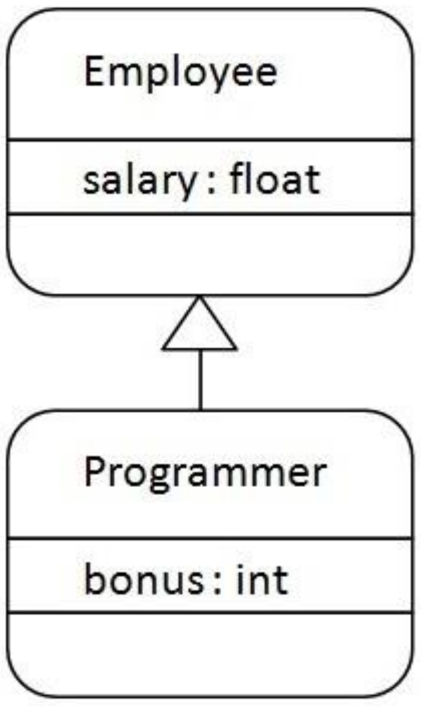
Why use inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- 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.

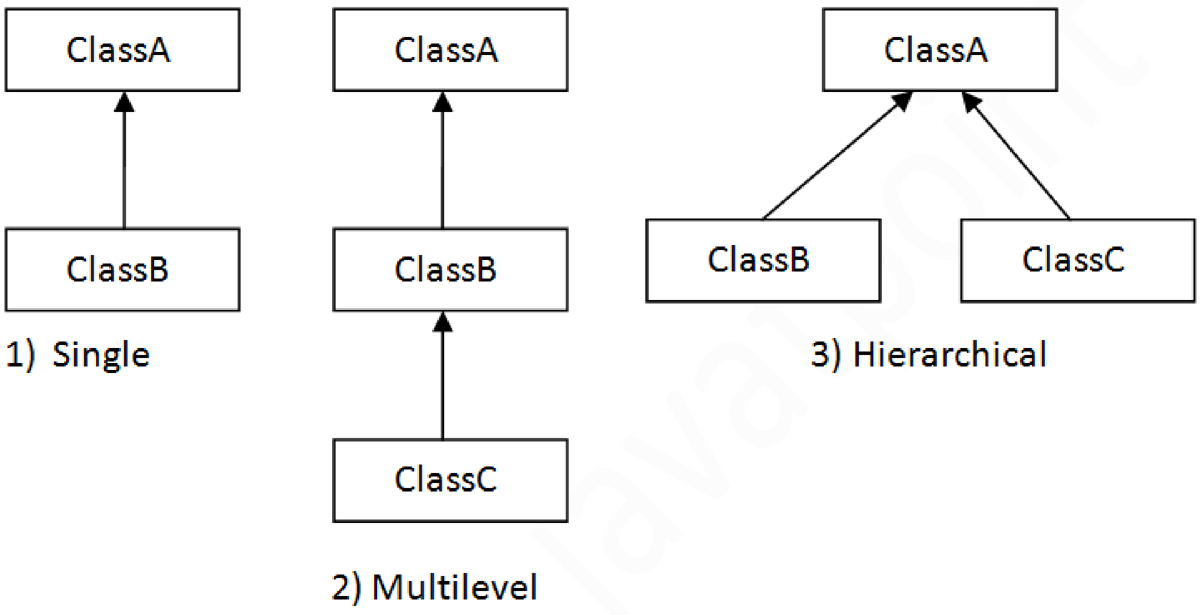
Java Inheritance Example



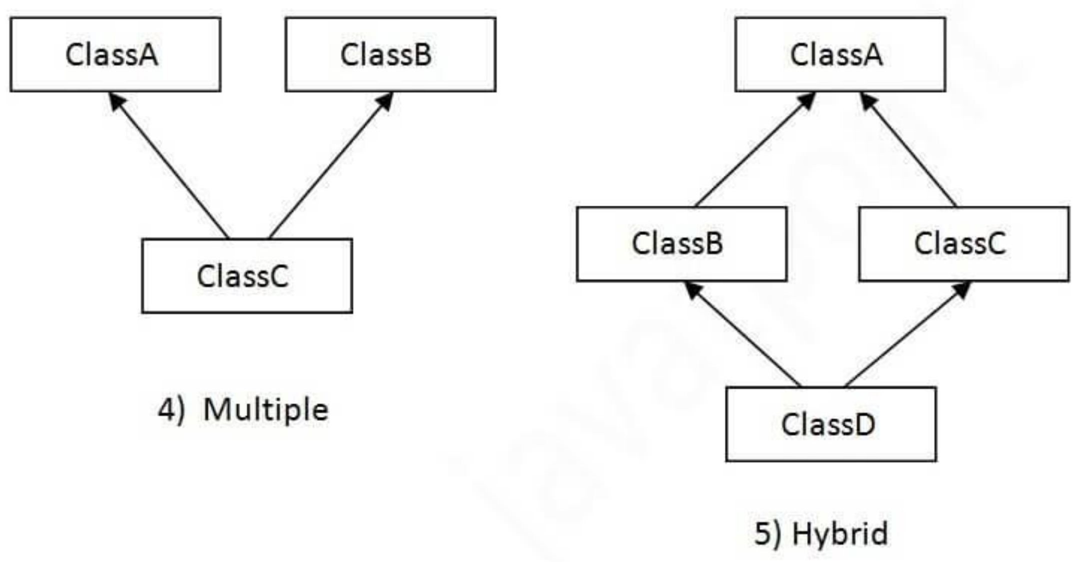
Types of inheritance in java

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.



When one class inherits multiple classes, it is known as multiple inheritance. For Example:



.

Single Inheritance Example

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.

File: TestInheritance.java

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

Output:

```
barking...
eating...
```

Multilevel Inheritance Example

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.

File: TestInheritance2.java

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

Output:

```
weeping...
barking...
eating...
```

Hierarchical Inheritance Example

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.

File: TestInheritance3.java

```java
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class Cat extends Animal{
void meow(){System.out.println("meowing...");}
}
class TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();
//c.bark();//C.T.Error
}}
```

Output:

```
meowing...
eating...
```

Multithreading and Exception Handling in JAVA

- Thread concept and life cycle of thread.
- Extending thread class and using thread methods
- Thread priority and runnable Interface
- Multithreading and Synchronization
- Exception Handling concepts and hierarchy
- Exception types and methods
- Concepts of “try, catch and throw and finally” in exceptions.
- User defined exceptions

Multithreading and Exception Handling in Java

In Java programming, threads enable concurrent execution and multitasking within an application. Understanding the life cycle of threads in Java and the various states is essential for efficient and synchronized thread management.

Importance of understanding the life cycle of Thread in Java and its states

The thread life cycle in Java is an important concept in multithreaded applications. Let’s see the importance of understanding life cycle of thread in java:

1. Understanding the life cycle of a thread in Java and the states of a thread is essential because it helps identify potential issues that can arise when creating or manipulating threads.
2. It allows developers to utilize resources more effectively and prevent errors related to multiple threads accessing shared data simultaneously.
3. Knowing the thread states in Java helps predict a program's behaviour and debug any issues that may arise.
4. It also guides the developer on properly suspending, resuming, and stopping a thread as required for a specific task.

The Life Cycle of Thread in Java - Threads State

In Java, the life cycle of Thread goes through various states. These states represent different stages of execution. Here are examples of each stage of the life cycle of Thread in Java with real-life use cases:

- **New (born) state**
    - Example: Creating a new thread using the Thread class constructor.
    - Use case: Creating a new thread to perform a background task while the main Thread continues with other operations
- **Runnable state**
    - Example: After calling the start() method on a thread, it enters the runnable state.
    - Use case: Multiple threads competing for CPU time to perform their tasks concurrently.

- **Running state**
    - Example: When a thread executes its code inside the run() method.
    - Use case: A thread executing a complex computation or performing a time-consuming task.
- **Blocked state:**
    - Example: When a thread tries to access a synchronized block or method, but another thread already holds the lock.
    - Use case: Multiple threads accessing a shared resource can only be obtained by a single Thread, such as a database or a file.
- **Waiting state:**
    - Example: Using the wait method inside a synchronized block, a thread can wait until another thread calls the notify() or notifyAll() methods to wake it up.
    - Use case: Implementing the producer-consumer pattern, where a thread waits for a specific condition to be met before continuing its execution
- **Timed waiting state:**
    - Example: Using methods like sleep(milliseconds) or join(milliseconds) causes a thread to enter the timed waiting state for the specified duration.
    - Use case: Adding delays between consecutive actions or waiting for the completion of other threads before proceeding.
- **Terminated state:**
    - Example: When the run() method finishes its execution or when the stop() method is called on the Thread.
    - Use case: Completing a task or explicitly stopping a thread's execution.

Example: Thread Life Cycle in Java:

Here's an example that demonstrates the life cycle of Thread in Java:

```
public class ThreadLifecycleExample {
    public static void main(String[] args) {
        Thread thread = new Thread(() -> {
            System.out.println("Thread is running.");
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
```

```
            e.printStackTrace();
        }
        System.out.println("Thread is terminating.");
    });
    System.out.println("Thread is in the New state.");
    thread.start();
    System.out.println("Thread is in the Runnable state.");

    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("Thread is in the Timed Waiting state.");
    try {
        thread.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("Thread is in the Terminated");
```

Handling Thread Exceptions and Terminating Threads

When dealing with exceptions in life cycle of a thread java, there are several ways to handle them.

- The most common way is wrapping the code in a try-catch block.
- Moreover, when a thread is no longer needed, it should be terminated for the application to avoid memory leaks and other issues. To do this, invoke the Thread's interrupt() or stop() methods. The former method sends an interruption signal to the Thread, while the latter stops it immediately and can cause instability in some cases.

Java Threads | How to create a thread in Java

There are two ways to create a thread:

1. By extending Thread class

2. By implementing Runnable interface.

Thread class:

Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r,String name)

Commonly used methods of Thread class:

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
3. **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public int getId():** returns the id of the thread.
12. **public Thread.State getState():** returns the state of the thread.
13. **public boolean isAlive():** tests if the thread is alive.
14. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. **public void suspend():** is used to suspend the thread(depricated).
16. **public void resume():** is used to resume the suspended thread(depricated).
17. **public void stop():** is used to stop the thread(depricated).

18. **public boolean isDaemon():** tests if the thread is a daemon thread.
19. **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
20. **public void interrupt():** interrupts the thread.
21. **public boolean isInterrupted():** tests if the thread has been interrupted.
22. **public static boolean interrupted():** tests if the current thread has been interrupted.

Runnable interface:

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

1. **public void run():** is used to perform action for a thread.

Starting a thread:

The **start() method** of Thread class is used to start a newly created thread. It performs the following tasks:

- A new thread starts(with new callstack).
- The thread moves from New state to the Runnable state.
- When the thread gets a chance to execute, its target run() method will run.

1) Java Thread Example by extending Thread class

FileName: Multi.java

```
class Multi extends Thread{
public void run(){
System.out.println("thread is running...");
}
public static void main(String args[]){
Multi t1=new Multi();
t1.start();
 }
}
```

Output:

```
thread is running...
```

2) Java Thread Example by implementing Runnable interface

FileName: Multi3.java

```java
class Multi3 implements Runnable{
public void run(){
System.out.println("thread is running...");
}

public static void main(String args[]){
Multi3 m1=new Multi3();
Thread t1 =new Thread(m1);   // Using the constructor Thread(Runnable r)
t1.start();
 }
}
```

Output:

```
thread is running...
```

If you are not extending the Thread class, your class object would not be treated as a thread object. So you need to explicitly create the Thread class object. We are passing the object of your class that implements Runnable so that your class run() method may execute.

3) Using the Thread Class: Thread(String Name)

We can directly use the Thread class to spawn new threads using the constructors defined above.

FileName: MyThread1.java

```java
public class MyThread1
{
// Main method
public static void main(String argvs[])
{
// creating an object of the Thread class using the constructor Thread(String name)
Thread t= new Thread("My first thread");

// the start() method moves the thread to the active state
t.start();
// getting the thread name by invoking the getName() method
String str = t.getName();
System.out.println(str);
}
}
```

Output:

```
My first thread
```

4) Using the Thread Class: Thread(Runnable r, String name)

Observe the following program.

FileName: MyThread2.java

```
public class MyThread2 implements Runnable
{
public void run()
{
System.out.println("Now the thread is running ...");
}

// main method
public static void main(String argvs[])
{
// creating an object of the class MyThread2
Runnable r1 = new MyThread2();

// creating an object of the class Thread using Thread(Runnable r, String name)
Thread th1 = new Thread(r1, "My new thread");

// the start() method moves the thread to the active state
th1.start();

// getting the thread name by invoking the getName() method
String str = th1.getName();
System.out.println(str);
}
}
```

Output:

```
My new thread
Now the thread is running ...
```

Java Runnable Interface

Java runnable is an interface used to execute code on a concurrent thread. It is an interface which is implemented by any class if we want that the instances of that class should be executed by a thread.

The runnable interface has an undefined method **run()** with void as return type, and it takes in no arguments. The method summary of the run() method is given below-

Method	Description
public void run()	This method takes in no arguments. When the object of a class implementing Runnable class is used to create a thread, then the run method is invoked in the thread which executes separately.

The runnable interface provides a standard set of rules for the instances of classes which wish to execute code when they are active. The most common use case of the Runnable interface is when we want only to override the run method. When a thread is started by the object of any class which is implementing Runnable, then it invokes the run method in the separately executing thread.

A class that implements Runnable runs on a different thread without subclassing Thread as it instantiates a Thread instance and passes itself in as the target. This becomes important as classes should not be subclassed unless there is an intention of modifying or enhancing the fundamental behavior of the class.

Runnable class is extensively used in network programming as each thread represents a separate flow of control. Also in multi-threaded programming, Runnable class is used. This interface is present in **java.lang** package.

Implementing Runnable

It is the easiest way to create a thread by implementing Runnable. One can create a thread on any object by implementing Runnable. To implement a Runnable, one has only to implement the run method.

public void run()

In this method, we have the code which we want to execute on a concurrent thread. In this method, we can use variables, instantiate classes, and perform an action like the same way the main thread does. The thread remains until the return of this method. The run method establishes an entry point to a new thread.

How to create a thread using Runnable interface

To create a thread using runnable, use the following code-

```
Runnable runnable = new MyRunnable();

Thread thread = new Thread(runnable);
thread.start();
```

The thread will execute the code which is mentioned in the run() method of the Runnable object passed in its argument.

Multithreading in Java

Multithreading in Java is a process of executing multiple threads simultaneously.

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation, etc.

Advantages of Java Multithreading

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.

2) You **can perform many operations together, so it saves time**.

3) Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

- Process-based Multitasking (Multiprocessing)
- Thread-based Multitasking (Multithreading)

1) Process-based Multitasking (Multiprocessing)

- Each process has an address in memory. In other words, each process allocates a separate memory area.
- A process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

2) Thread-based Multitasking (Multithreading)

- Threads share the same address space.
- A thread is lightweight.
- Cost of communication between the thread is low.

Synchronization in Java

Synchronization in Java is the capability to control the access of multiple threads to any shared resource.

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

Why use Synchronization?

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Here, we will discuss only thread synchronization.

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
    1. Synchronized method.
    2. Synchronized block.
    3. Static synchronization.
2. Cooperation (Inter-thread communication in java)

Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. It can be achieved by using the following three ways:

1. By Using Synchronized Method
2. By Using Synchronized Block
3. By Using Static Synchronization

Exception Handling in Java

- Exception Handling

- Advantage of Exception Handling
- Hierarchy of Exception classes
- Types of Exception
- Exception Example
- Scenarios where an exception may occur

The **Exception Handling in Java** is one of the powerful *mechanism to handle the runtime errors* so that the normal flow of the application can be maintained.

In this tutorial, we will learn about Java exceptions, it's types, and the difference between checked and unchecked exceptions.

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.

What is Exception Handling?

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Advantage of Exception Handling

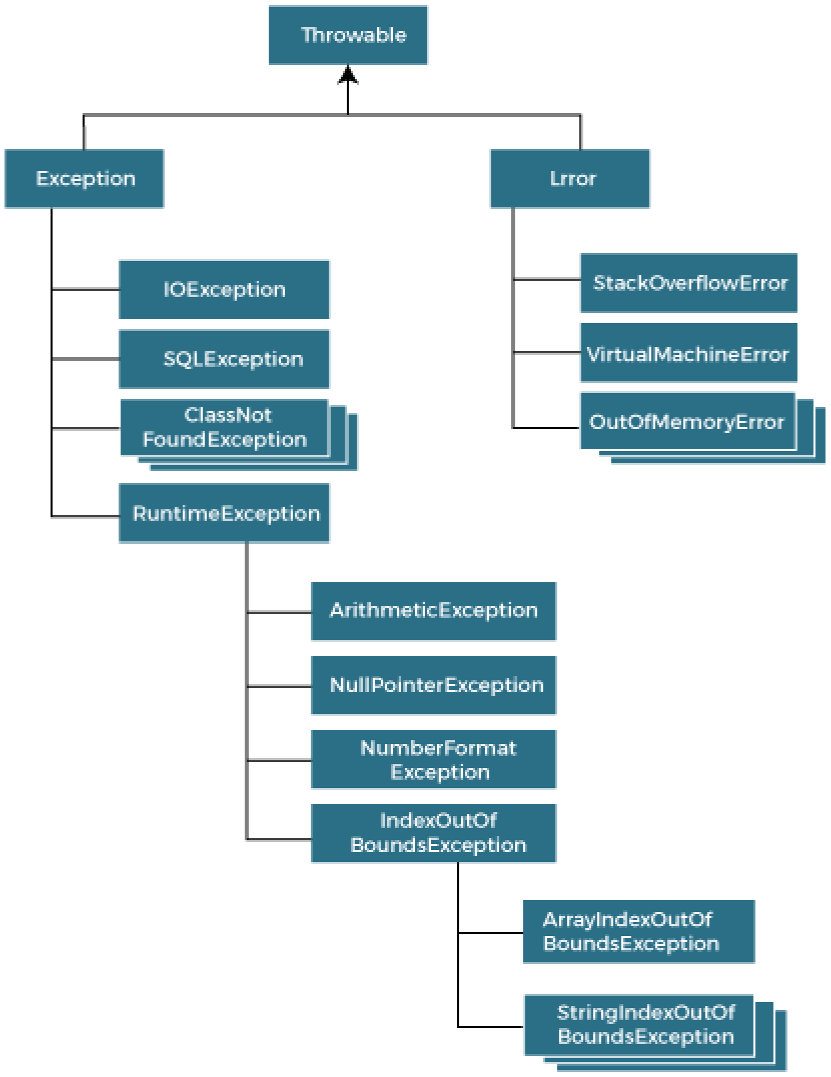
The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions. Let's consider a scenario:

```
statement 1;
statement 2;
statement 3;
statement 4;
statement 5;//exception occurs
statement 6;
statement 7;
statement 8;
statement 9;
statement 10;
```

Suppose there are 10 statements in a Java program and an exception occurs at statement 5; the rest of the code will not be executed, i.e., statements 6 to 10 will not be executed. However, when we perform exception handling, the rest of the statements will be executed. That is why we use exception handling in Java.

Hierarchy of Java Exception classes

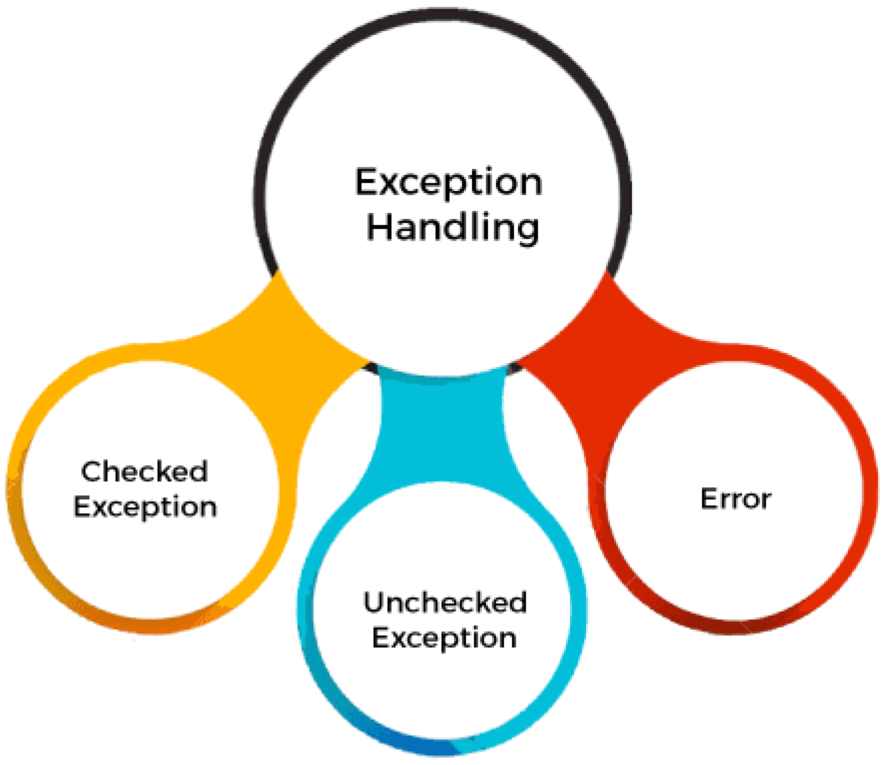
The java.lang.Throwable class is the root class of Java Exception hierarchy inherited by two subclasses: Exception and Error. The hierarchy of Java Exception classes is given below:



Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

1. Checked Exception
2. Unchecked Exception
3. Error



Difference between Checked and Unchecked Exceptions

1) Checked Exception

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

Java Exception Keywords

Java provides five keywords that are used to handle the exception. The following table describes each.

Keyword	**Description**

try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
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 necessary 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 specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

Java Exception Handling Example

Let's see an example of Java Exception Handling in which we are using a try-catch statement to handle the exception.

JavaExceptionExample.java

```java
public class JavaExceptionExample{
  public static void main(String args[]){
   try{
      //code that may raise exception
      int data=100/0;
   }catch(ArithmeticException e){System.out.println(e);}
   //rest code of the program
   System.out.println("rest of the code...");
  }
}
```

Test it Now

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...
```

In the above example, 100/0 raises an ArithmeticException which is handled by a try-catch block.

Common Scenarios of Java Exceptions

There are given some scenarios where unchecked exceptions may occur. They are as follows:

1) A scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

```java
int a=50/0;//ArithmeticException
```

2) A scenario where NullPointerException occurs

If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

```java
String s=null;
System.out.println(s.length());//NullPointerException
```

3) A scenario where NumberFormatException occurs

If the formatting of any variable or number is mismatched, it may result into NumberFormatException. Suppose we have a string variable that has characters; converting this variable into digit will cause NumberFormatException.

```java
String s="abc";
int i=Integer.parseInt(s);//NumberFormatException
```

4) A scenario where ArrayIndexOutOfBoundsException occurs

When an array exceeds to it's size, the ArrayIndexOutOfBoundsException occurs. there may be other reasons to occur ArrayIndexOutOfBoundsException. Consider the following statements.

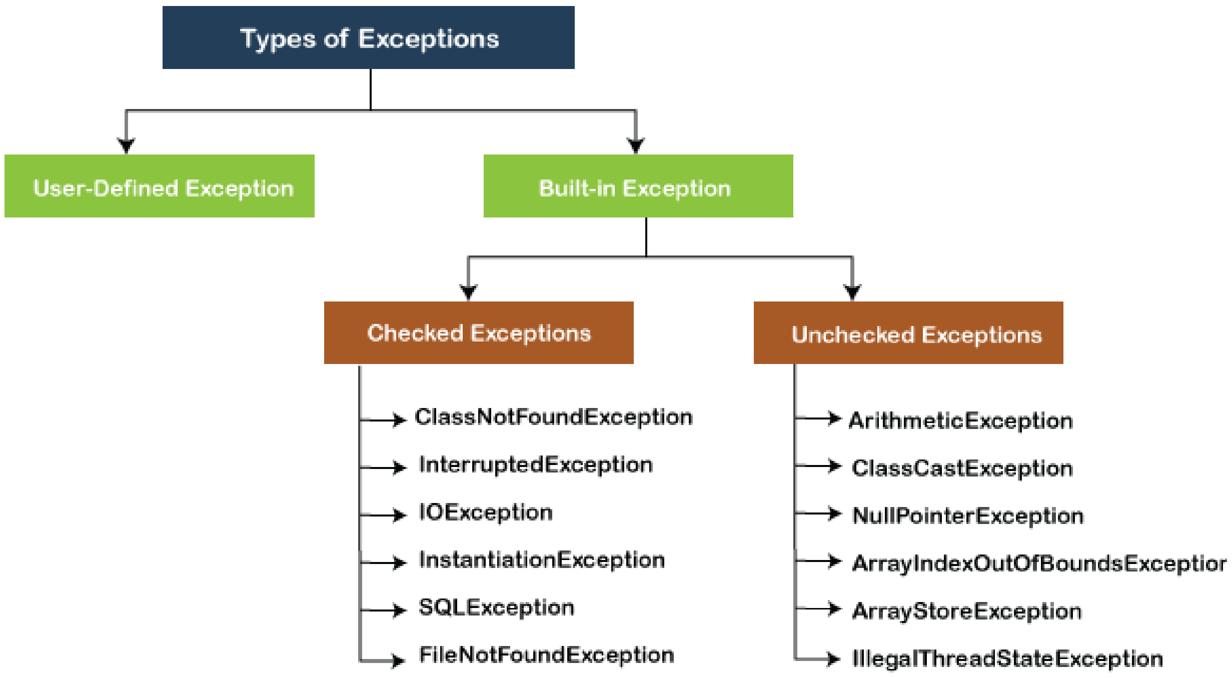
```java
int a[]=new int[5];
a[10]=50; //ArrayIndexOutOfBoundsException
```

Types of Exception in Java

In Java, **exception** is an event that occurs during the execution of a program and disrupts the normal flow of the program's instructions. Bugs or errors that we don't want and restrict our program's normal execution of code are referred to as **exceptions**. In this section, we will focus on the **types of exceptions in Java** and the differences between the two.

Exceptions can be categorized into two ways:

1. Built-in Exceptions
   - Checked Exception
   - Unchecked Exception
2. User-Defined Exceptions



Built-in Exception

Exceptions that are already available in **Java libraries** are referred to as **built-in exception**. These exceptions are able to define the error situation so that we can understand the reason of getting this error. It can be categorized into two broad categories, i.e., **checked exceptions** and **unchecked exception**.

Checked Exception

Checked exceptions are called **compile-time** exceptions because these exceptions are checked at compile-time by the compiler. The compiler ensures whether the programmer handles the exception or not. The programmer should have to handle the exception; otherwise, the system has shown a compilation error.

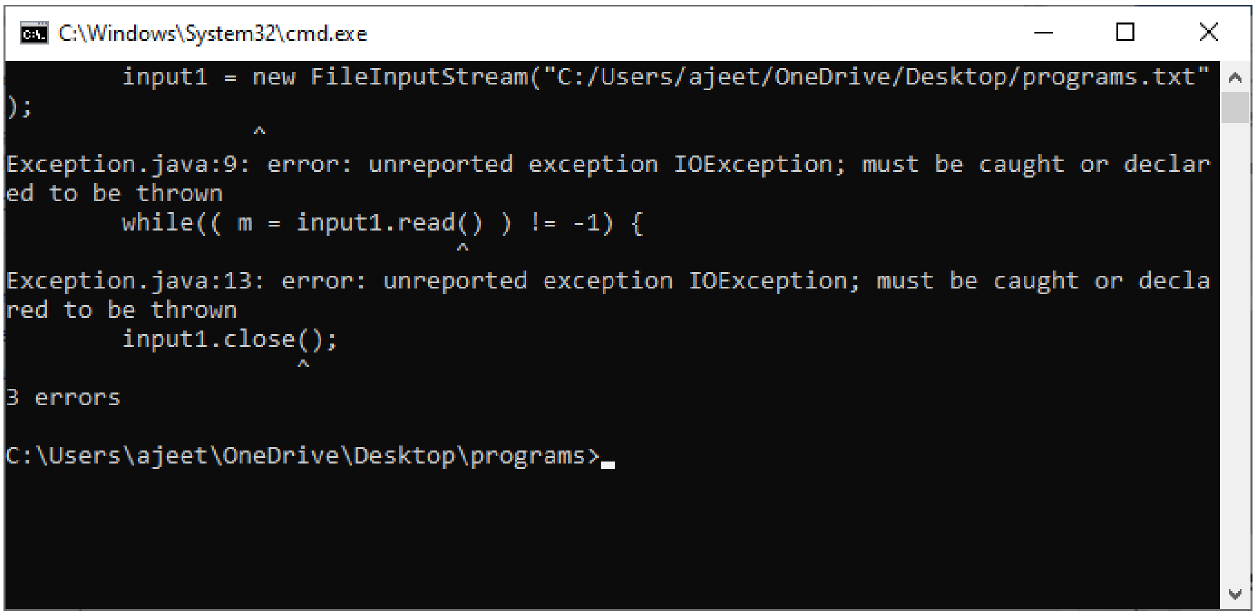
CheckedExceptionExample.java

```java
import java.io.*;
class CheckedExceptionExample {
    public static void main(String args[]) {
        FileInputStream file_data = null;
        file_data = new FileInputStream("C:/Users/ajeet/OneDrive/Desktop/Hello.txt");
        int m;
        while(( m = file_data.read() ) != -1) {
            System.out.print((char)m);
        }
        file_data.close();
    }
}
```

In the above code, we are trying to read the **Hello.txt** file and display its data or content on the screen. The program throws the following exceptions:

1. The **FileInputStream(File filename)** constructor throws the **FileNotFoundException** that is checked exception.
2. The **read()** method of the **FileInputStream** class throws the **IOException**.
3. The **close()** method also throws the IOException.

Output:



```
        input1 = new FileInputStream("C:/Users/ajeet/OneDrive/Desktop/programs.txt"
);
                 ^
Exception.java:9: error: unreported exception IOException; must be caught or declar
ed to be thrown
        while(( m = input1.read() ) != -1) {
                               ^
Exception.java:13: error: unreported exception IOException; must be caught or decla
red to be thrown
        input1.close();
                    ^
3 errors

C:\Users\ajeet\OneDrive\Desktop\programs>
```

How to resolve the error?

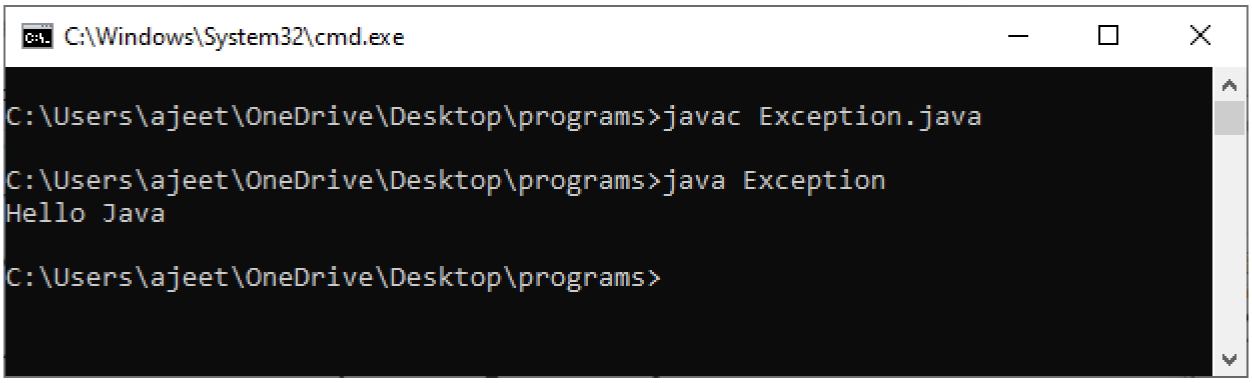
There are basically two ways through which we can solve these errors.

1) The exceptions occur in the main method. We can get rid from these compilation errors by declaring the exception in the main method using **the throws** We only declare the IOException, not FileNotFoundException, because of the child-parent relationship. The IOException class is the parent class of FileNotFoundException, so this exception will automatically cover by IOException. We will declare the exception in the following way:

```
class Exception{
    public static void main(String args[])  throws IOException {
        ...
        ...
}
```

If we compile and run the code, the errors will disappear, and we will see the data of the file.

ADVERTISEM



C:\Windows\System32\cmd.exe

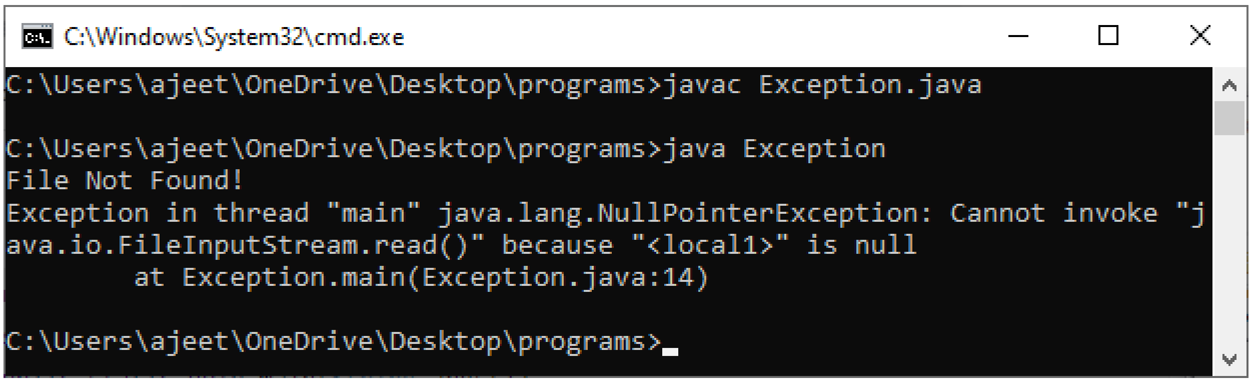
```
C:\Users\ajeet\OneDrive\Desktop\programs>javac Exception.java

C:\Users\ajeet\OneDrive\Desktop\programs>java Exception
Hello Java

C:\Users\ajeet\OneDrive\Desktop\programs>
```

2) We can also handle these exception using **try-catch** However, the way which we have used above is not correct. We have to a give meaningful message for each exception type. By doing that it would be easy to understand the error. We will use the try-catch block in the following way:

Exception.java

```
import java.io.*;
class Exception{
    public static void main(String args[]) {
        FileInputStream file_data = null;
        try{
            file_data = new FileInputStream("C:/Users/ajeet/OneDrive/Desktop/programs/Hell.txt");
        }catch(FileNotFoundException fnfe){
            System.out.println("File Not Found!");
        }
        int m;
        try{
            while(( m = file_data.read() ) != -1) {
                System.out.print((char)m);
            }
            file_data.close();
        }catch(IOException ioe){
            System.out.println("I/O error occurred: "+ioe);
        }
    }
}
```

We will see a proper error message **"File Not Found!"** on the console because there is no such file in that location.



```
C:\Users\ajeet\OneDrive\Desktop\programs>javac Exception.java

C:\Users\ajeet\OneDrive\Desktop\programs>java Exception
File Not Found!
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "java.io.FileInputStream.read()" because "<local1>" is null
        at Exception.main(Exception.java:14)

C:\Users\ajeet\OneDrive\Desktop\programs>
```

Unchecked Exceptions

The **unchecked** exceptions are just opposite to the **checked** exceptions. The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception, and even if we didn't handle or declare it, the program would not give a compilation error. Usually, it occurs when the user provides bad data during the interaction with the program.

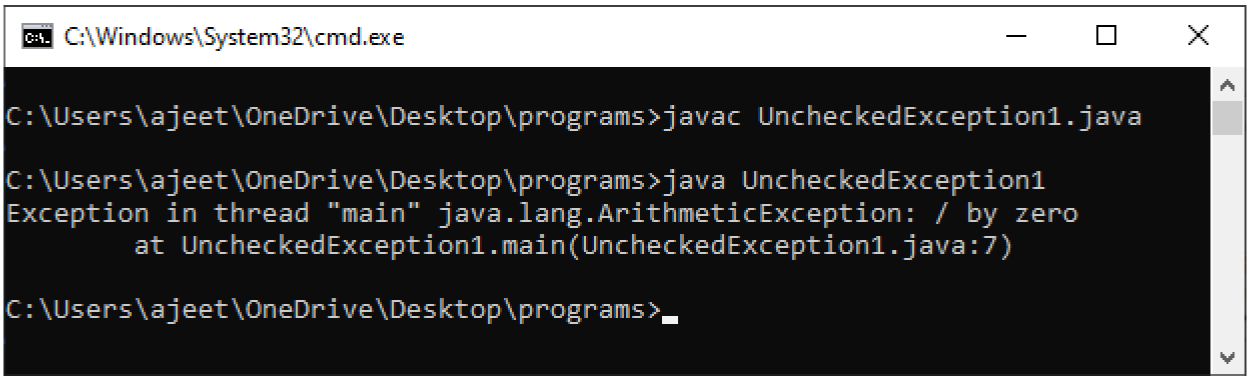
Note: The RuntimeException class is able to resolve all the unchecked exceptions because of the child-parent relationship.

UncheckedExceptionExample1.java

```java
class UncheckedExceptionExample1 {
  public static void main(String args[])
  {
   int postive = 35;
   int zero = 0;
   int result = positive/zero;
   //Give Unchecked Exception here.
System.out.println(result);
  }
}
```

In the above program, we have divided 35 by 0. The code would be compiled successfully, but it will throw an ArithmeticException error at runtime. On dividing a number by 0 throws the divide by zero exception that is a uncheck exception.

Output:

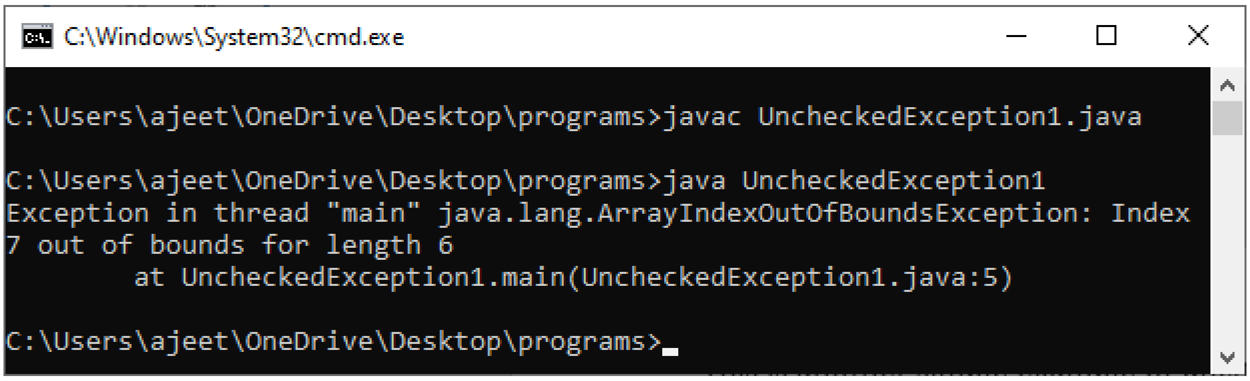


```
C:\Users\ajeet\OneDrive\Desktop\programs>javac UncheckedException1.java

C:\Users\ajeet\OneDrive\Desktop\programs>java UncheckedException1
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at UncheckedException1.main(UncheckedException1.java:7)

C:\Users\ajeet\OneDrive\Desktop\programs>_
```

UncheckedException1.java

```java
class UncheckedException1 {
  public static void main(String args[])
  {
   int num[] ={10,20,30,40,50,60};
   System.out.println(num[7]);
  }
}
```

Output:



```
C:\Users\ajeet\OneDrive\Desktop\programs>javac UncheckedException1.java

C:\Users\ajeet\OneDrive\Desktop\programs>java UncheckedException1
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index
7 out of bounds for length 6
        at UncheckedException1.main(UncheckedException1.java:5)

C:\Users\ajeet\OneDrive\Desktop\programs>_
```

In the above code, we are trying to get the element located at position 7, but the length of the array is 6. The code compiles successfully, but throws the ArrayIndexOutOfBoundsException at runtime

User-defined Exception

In Java, we already have some built-in exception classes like **ArrayIndexOutOfBoundsException**, **NullPointerException**, and **ArithmeticException**. These exceptions are restricted to trigger on some predefined conditions. In Java, we can write our own exception class by extends the Exception class. We can throw our own exception on a particular condition using the throw keyword. For creating a user-defined exception, we should have basic knowledge of **the try-catch** block and **throw** keyword.

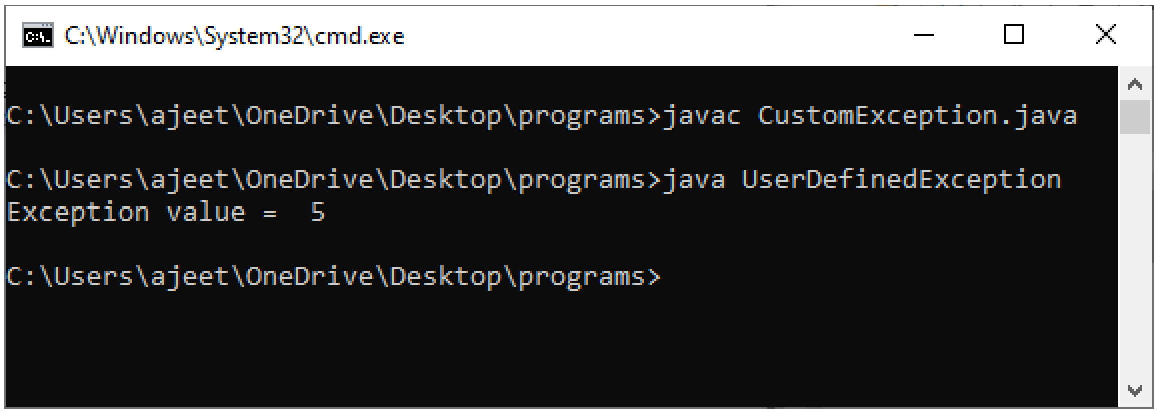
Let's write a Java program and create user-defined exception.

UserDefinedException.java

```java
import java.util.*;
class UserDefinedException{
    public static void main(String args[]){
        try{
            throw new NewException(5);
        }
        catch(NewException ex){
            System.out.println(ex) ;
        }
    }
}
```

```
class NewException extends Exception{
    int x;
    NewException(int y) {
        x=y;
    }
    public String toString(){
        return ("Exception value =  "+x) ;
    }
}
```

Output:



```
C:\Users\ajeet\OneDrive\Desktop\programs>javac CustomException.java

C:\Users\ajeet\OneDrive\Desktop\programs>java UserDefinedException
Exception value =  5

C:\Users\ajeet\OneDrive\Desktop\programs>
```

Description:

In the above code, we have created two classes, i.e., **UserDefinedException** and **NewException**. The **UserDefinedException** has our main method, and the **NewException** class is our user-defined exception class, which extends **exception**. In the **NewException** class, we create a variable **x** of type integer and assign a value to it in the constructor. After assigning a value to that variable, we return the exception message.

In the **UserDefinedException** class, we have added a **try-catch** block. In the try section, we throw the exception, i.e., **NewException** and pass an integer to it. The value will be passed to the NewException class and return a message. We catch that message in the catch block and show it on the screen.

Difference Between Checked and Unchecked Exception

S.No	Checked Exception	Unchecked Exception

1.	These exceptions are checked at compile time. These exceptions are handled at compile time too.	These exceptions are just opposite to the checked exceptions. These exceptions are not checked and handled at compile time.
2.	These exceptions are direct subclasses of exception but not extended from RuntimeException class.	They are the direct subclasses of the RuntimeException class.
3.	The code gives a compilation error in the case when a method throws a checked exception. The compiler is not able to handle the exception on its own.	The code compiles without any error because the exceptions escape the notice of the compiler. These exceptions are the results of user-created errors in programming logic.
4.	These exceptions mostly occur when the probability of failure is too high.	These exceptions occur mostly due to programming mistakes.
5.	Common checked exceptions include IOException, DataAccessException, InterruptedException, etc.	Common unchecked exceptions include ArithmeticException, InvalidClassException, NullPointerException, etc.
6.	These exceptions are propagated using the throws keyword.	These are automatically propagated.
7.	It is required to provide the try-catch and try-finally block to handle the checked exception.	In the case of unchecked exception it is not mandatory.

Bugs or errors that we don't want and restrict the normal execution of the programs are referred to as **exceptions**.

ArithmeticException, ArrayIndexOutOfBoundExceptions, ClassNotFoundExceptions etc. are come in the category of **Built-in Exception**. Sometimes, the built-in exceptions are not sufficient to explain or describe certain situations. For describing these situations, we have to create our own exceptions by creating an exception class as a subclass of the **Exception** class. These types of exceptions come in the category of **User-Defined Exception**.

Java try-catch block

Java try block

Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.

If an exception occurs at the particular statement in the try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception.

Java try block must be followed by either catch or finally block.

Syntax of Java try-catch

```
try{
//code that may throw an exception
}catch(Exception_class_Name ref){}
```

Syntax of try-finally block

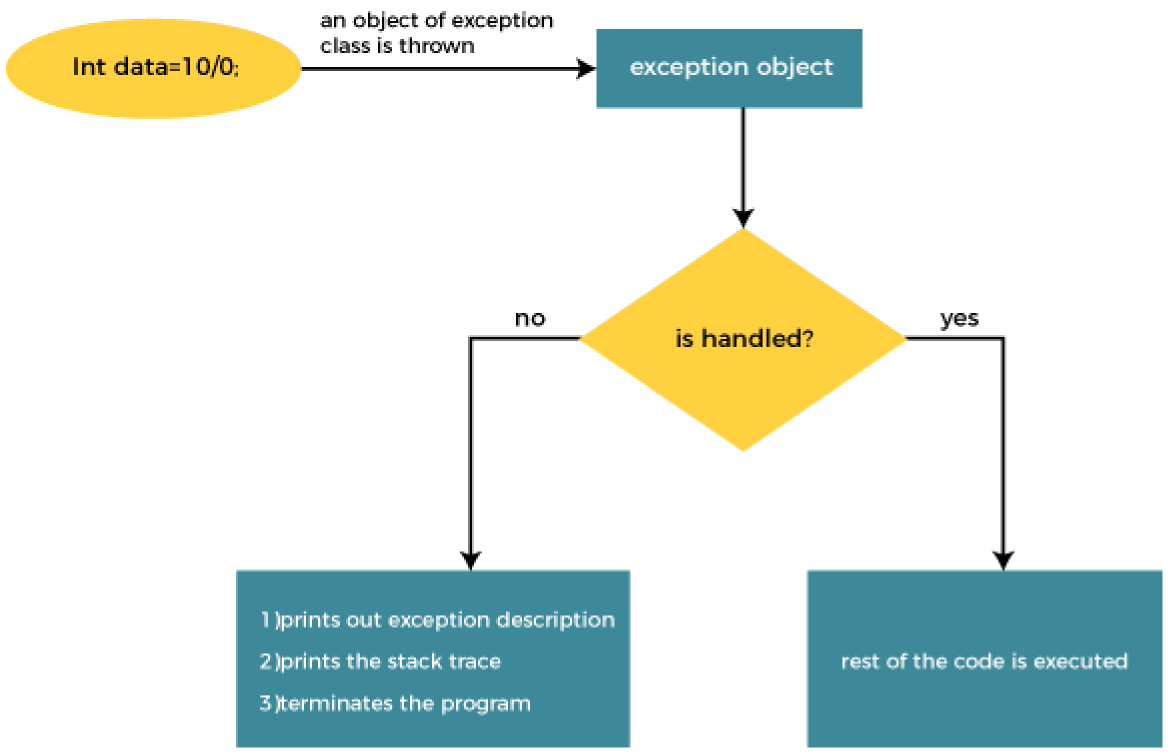
```
try{
//code that may throw an exception
}finally{}
```

Java catch block

Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

The catch block must be used after the try block only. You can use multiple catch block with a single try block.

Internal Working of Java try-catch block


Int data=10/0;
an object of exception class is thrown
exception object
is handled?
no
yes
1)prints out exception description
2)prints the stack trace
3)terminates the program
rest of the code is executed

The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

- Prints out exception description.
- Prints the stack trace (Hierarchy of methods where the exception occurred).
- Causes the program to terminate.

But if the application programmer handles the exception, the normal flow of the application is maintained, i.e., rest of the code is executed.

Problem without exception handling

Let's try to understand the problem if we don't use a try-catch block.

Example 1

TryCatchExample1.java

```java
public class TryCatchExample1 {

    public static void main(String[] args) {

        int data=50/0; //may throw exception

        System.out.println("rest of the code");

    }

}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

As displayed in the above example, the **rest of the code** is not executed (in such case, the **rest of the code** statement is not printed).

There might be 100 lines of code after the exception. If the exception is not handled, all the code below the exception won't be executed.

Solution by exception handling

Let's see the solution of the above problem by a java try-catch block.

Example 2

TryCatchExample2.java

```
public class TryCatchExample2 {

    public static void main(String[] args) {
        try
        {
        int data=50/0; //may throw exception
        }
            //handling the exception
        catch(ArithmeticException e)
        {
            System.out.println(e);
        }
        System.out.println("rest of the code");
    }

}
```

Output:

```
java.lang.ArithmeticException: / by zero
rest of the code
```

As displayed in the above example, the **rest of the code** is executed, i.e., the **rest of the code** statement is printed.

Example 3

In this example, we also kept the code in a try block that will not throw an exception.

TryCatchExample3.java

```java
public class TryCatchExample3 {

    public static void main(String[] args) {
        try
        {
        int data=50/0; //may throw exception
                          // if exception occurs, the remaining statement will not exceute
        System.out.println("rest of the code");
        }
            // handling the exception
        catch(ArithmeticException e)
        {
            System.out.println(e);
        }

    }

}
```

Output:

```
java.lang.ArithmeticException: / by zero
```

Here, we can see that if an exception occurs in the try block, the rest of the block code will not execute.

Example 4

Here, we handle the exception using the parent class exception.

TryCatchExample4.java

```java
public class TryCatchExample4 {

    public static void main(String[] args) {
        try
        {
        int data=50/0; //may throw exception
        }
            // handling the exception by using Exception class
        catch(Exception e)
        {
            System.out.println(e);
        }
        System.out.println("rest of the code");
    }

}
```

Output:

```
java.lang.ArithmeticException: / by zero
rest of the code
```

Java throw Exception

In Java, exceptions allows us to write good quality codes where the errors are checked at the compile time instead of runtime and we can create custom exceptions making the code recovery and debugging easier.

Java throw keyword

The Java throw keyword is used to throw an exception explicitly.

We specify the **exception** object which is to be thrown. The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.
We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception. We will discuss custom exceptions later in this section.
We can also define our own set of conditions and throw an exception explicitly using throw keyword. For example, we can throw ArithmeticException if we divide a number by another number. Here, we just need to set the condition and throw exception using throw keyword.
The syntax of the Java throw keyword is given below.
throw Instance i.e.,

```
throw new exception_class("error message");
```

Let's see the example of throw IOException.

```
throw new IOException("sorry device error");
```

Where the Instance must be of type Throwable or subclass of Throwable. For example, Exception is the sub class of Throwable and the user-defined exceptions usually extend the Exception class.

Java throw keyword Example

Example 1: Throwing Unchecked Exception

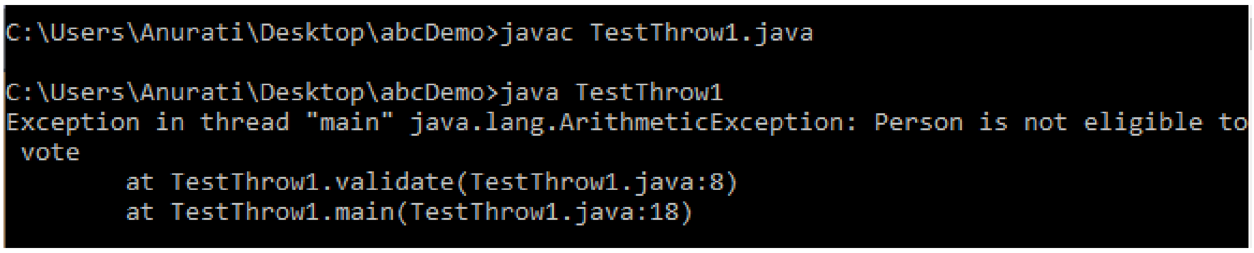
In this example, we have created a method named validate() that accepts an integer as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

TestThrow1.java

In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

```
public class TestThrow1 {
    //function to check if person is eligible to vote or not
    public static void validate(int age) {
        if(age<18) {
            //throw Arithmetic exception if not eligible to vote
            throw new ArithmeticException("Person is not eligible to vote");
        }
        else {
            System.out.println("Person is eligible to vote!!");
        }
    }
    //main method
    public static void main(String args[]){
        //calling the function
        validate(13);
        System.out.println("rest of the code...");
  }
}
```

Output:



```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow1.java

C:\Users\Anurati\Desktop\abcDemo>java TestThrow1
Exception in thread "main" java.lang.ArithmeticException: Person is not eligible to
 vote
        at TestThrow1.validate(TestThrow1.java:8)
        at TestThrow1.main(TestThrow1.java:18)
```

The above code throw an unchecked exception. Similarly, we can also throw unchecked and user defined exceptions.

Note: If we throw unchecked exception from a method, it is must to handle the exception or declare in throws clause.

If we throw a checked exception using throw keyword, it is must to handle the exception using catch block or the method must declare it using throws declaration.

Example 2: Throwing Checked Exception

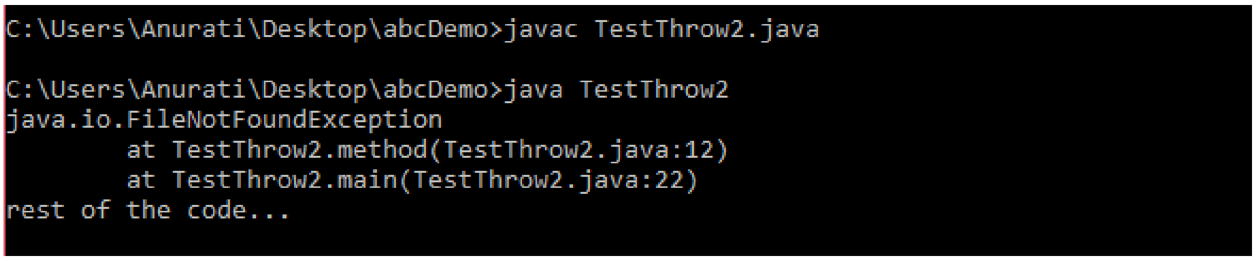
Note: Every subclass of Error and RuntimeException is an unchecked exception in Java. A checked exception is everything else under the Throwable class.

TestThrow2.java

```
import java.io.*;

```

```java
public class TestThrow2 {

    //function to check if person is eligible to vote or not
    public static void method() throws FileNotFoundException {

        FileReader file = new FileReader("C:\\Users\\Anurati\\Desktop\\abc.txt");
        BufferedReader fileInput = new BufferedReader(file);


        throw new FileNotFoundException();

    }
    //main method
    public static void main(String args[]){
        try
        {
            method();
        }
        catch (FileNotFoundException e)
        {
            e.printStackTrace();
        }
        System.out.println("rest of the code...");
  }
}
```

Output:



```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow2.java

C:\Users\Anurati\Desktop\abcDemo>java TestThrow2
java.io.FileNotFoundException
        at TestThrow2.method(TestThrow2.java:12)
        at TestThrow2.main(TestThrow2.java:22)
rest of the code...
```

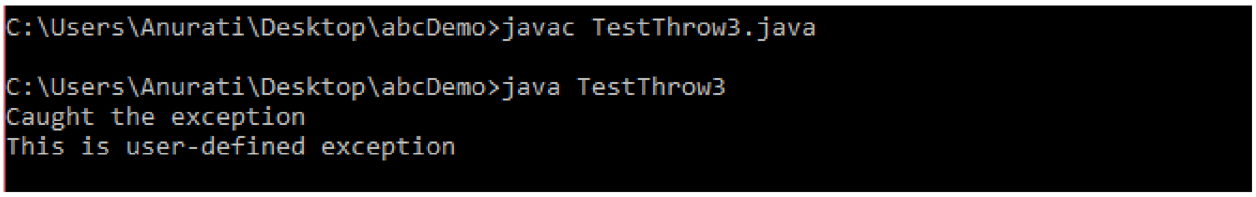
Example 3: Throwing User-defined Exception

exception is everything else under the Throwable class.

TestThrow3.java

```java
// class represents user-defined exception
class UserDefinedException extends Exception
{
    public UserDefinedException(String str)
    {
        // Calling constructor of parent Exception
        super(str);
    }
}
// Class that uses above MyException
public class TestThrow3
{
    public static void main(String args[])
    {
        try
        {
            // throw an object of user defined exception
            throw new UserDefinedException("This is user-defined exception");
        }
        catch (UserDefinedException ude)
        {
            System.out.println("Caught the exception");
            // Print the message from MyException object
            System.out.println(ude.getMessage());
        }
    }
}
```

Output:



```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow3.java

C:\Users\Anurati\Desktop\abcDemo>java TestThrow3
Caught the exception
This is user-defined exception
```

Difference between final, finally and finalize

The final, finally, and finalize are keywords in Java that are used in exception handling. Each of these keywords has a different functionality. The basic difference between final, finally and finalize is that the final is an access modifier, finally is the block in Exception Handling and finalize is the method of object class.

Along with this, there are many differences between final, finally and finalize. A list of differences between final, finally and finalize are given below:

Sr. no.	Key	final	finally	finalize
1.	**Definition**	**final is the keyword and access modifier which is used to apply restrictions on a class, method or variable.**	**finally is the block in Java Exception Handling to execute the important code whether the exception occurs or not.**	**finalize is the method in Java which is used to perform clean up processing just before object is garbage collected.**
2.	**Applicable to**	**Final keyword is used with the classes, methods and variables.**	**Finally block is always related to the try and catch block in exception handling.**	**finalize() method is used with the objects.**
3.	**Functionality**	**(1) Once declared, final variable becomes constant and cannot be modified. (2) final method cannot be**	**(1) finally block runs the important code even if exception occurs or not. (2) finally block cleans up all the**	**finalize method performs the cleaning activities with respect to the object before its destruction.**

		overridden by sub class. (3) final class cannot be inherited.	resources used in try block	
4.	Execution	Final method is executed only when we call it.	Finally block is executed as soon as the try-catch block is executed. It's execution is not dependant on the exception.	finalize method is executed just before the object is destroyed.

Java final Example

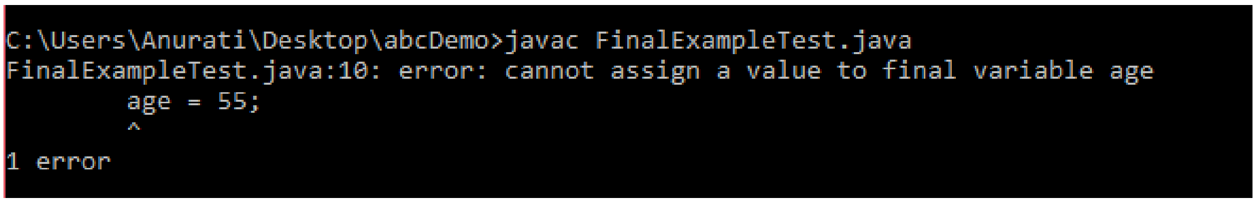
Let's consider the following example where we declare final variable age. Once declared it cannot be modified.

FinalExampleTest.java

```java
public class FinalExampleTest {
    //declaring final variable
    final int age = 18;
    void display() {

    // reassigning value to age variable
    // gives compile time error
    age = 55;
    }

    public static void main(String[] args) {

```

```
    FinalExampleTest obj = new FinalExampleTest();
    // gives compile time error
    obj.display();
    }
}
```

Output:



```
C:\Users\Anurati\Desktop\abcDemo>javac FinalExampleTest.java
FinalExampleTest.java:10: error: cannot assign a value to final variable age
        age = 55;
        ^
1 error
```

In the above example, we have declared a variable final. Similarly, we can declare the methods and classes final using the final keyword.

Java finally Example

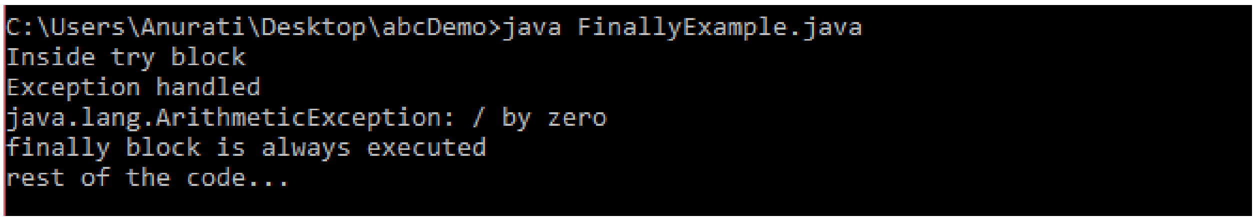
Let's see the below example where the Java code throws an exception and the catch block handles that exception. Later the finally block is executed after the try-catch block. Further, the rest of the code is also executed normally.

FinallyExample.java

```
public class FinallyExample {
      public static void main(String args[]){
      try {
        System.out.println("Inside try block");
      // below code throws divide by zero exception
       int data=25/0;
       System.out.println(data);
      }
      // handles the Arithmetic Exception / Divide by zero exception
      catch (ArithmeticException e){
        System.out.println("Exception handled");
        System.out.println(e);
      }
      // executes regardless of exception occurred or not
      finally {
```

```
      System.out.println("finally block is always executed");
    }
    System.out.println("rest of the code...");
    }
  }
```

Output:



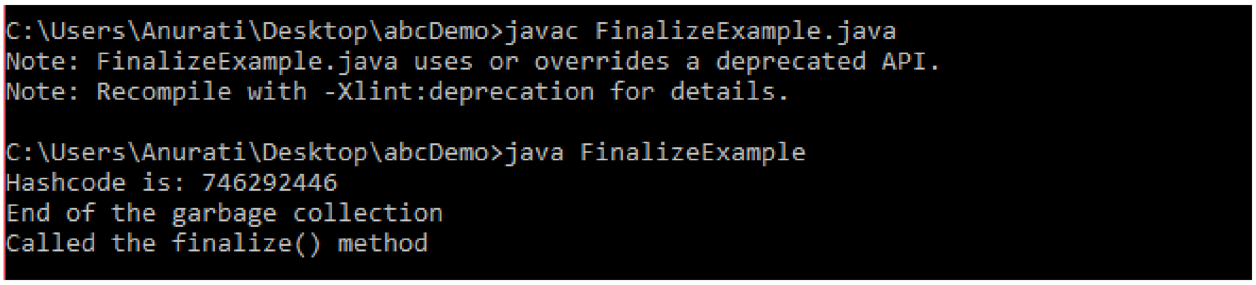
```
C:\Users\Anurati\Desktop\abcDemo>java FinallyExample.java
Inside try block
Exception handled
java.lang.ArithmeticException: / by zero
finally block is always executed
rest of the code...
```

Java finalize Example

FinalizeExample.java

```
public class FinalizeExample {

   public static void main(String[] args)
  {
      FinalizeExample obj = new FinalizeExample();
      // printing the hashcode
      System.out.println("Hashcode is: " + obj.hashCode());
      obj = null;
      // calling the garbage collector using gc()
      System.gc();
      System.out.println("End of the garbage collection");
  }
 // defining the finalize method
  protected void finalize()
  {
      System.out.println("Called the finalize() method");
  }
}
```

Output:



```
C:\Users\Anurati\Desktop\abcDemo>javac FinalizeExample.java
Note: FinalizeExample.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.

C:\Users\Anurati\Desktop\abcDemo>java FinalizeExample
Hashcode is: 746292446
End of the garbage collection
Called the finalize() method
```

Abstract Classes and Interfaces in JAVA

- Concept of Virtual methods.
- Concept of Abstract classes and methods
- Features of Abstract Classes
- JAVA Interfaces and their advantages
- Method Overriding in JAVA
- Polymorphism in JAVA
- Creating , implementing and extending interfaces
- Creating and using Packages in JAVA.

Concept of virtual function

Virtual Function in Java

A virtual function or virtual method in an OOP language is a function or method used to override the behavior of the function in an inherited class with the same signature to achieve the polymorphism.

When the programmers switch the technology from C++ to Java, they think about where is the virtual function in Java. In C++, the virtual function is defined using the virtual keyword, but in Java, it is achieved using different techniques. See Virtual function in C++.

Java is an object-oriented programming language; it supports OOPs features such as polymorphism, abstraction, inheritance, etc. These concepts are based on objects, classes, and member functions.

How to use Virtual function in Java

The virtual keyword is not used in Java to define the virtual function; instead, the virtual functions and methods are achieved using the following techniques:

- We can override the virtual function with the inheriting class function using the same function name. Generally, the virtual function is defined in the parent class and override it in the inherited class.
- The virtual function is supposed to be defined in the derived class. We can call it by referring to the derived class's object using the reference or pointer of the base class.
- A virtual function should have the same name and parameters in the base and derived class.
- For the virtual function, an IS-A relationship is necessary, which is used to define the class hierarchy in inheritance.
- The Virtual function cannot be private, as the private functions cannot be overridden.

- A virtual function or method also cannot be final, as the final methods also cannot be overridden.
- Static functions are also cannot be overridden; so, a virtual function should not be static.

Examples:-

<u>Parent.Java</u>:

```
class Parent {
void v1() //Declaring function
{
System.out.println("Inside the Parent Class");
}
}
```

<u>Child.java</u>:

```
public class Child extends Parent{
        void v1() // Overriding function from the Parent class
        {
        System.out.println("Inside the Child Class");
        }
        public static void main(String args[]){
        Parent ob1 = new Child(); //Refering the child class object using the parent class
        ob1.v1();
        }
        }
```

Output:

Inside the Child Class

Concept of Abstract classes and methods

Data abstraction is the process of hiding certain details and showing only essential information to the user.
Abstraction can be achieved with either abstract classes or interfaces (which you will learn more about in the next chapter).

The abstract keyword is a non-access modifier, used for classes and methods:

- Abstract class: is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
- Abstract method: can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

An abstract class can have both abstract and regular methods:

```
abstract class Animal {
  public abstract void animalSound();
  public void sleep() {
    System.out.println("Zzz");
  }
}
```

From the example above, it is not possible to create an object of the Animal class:

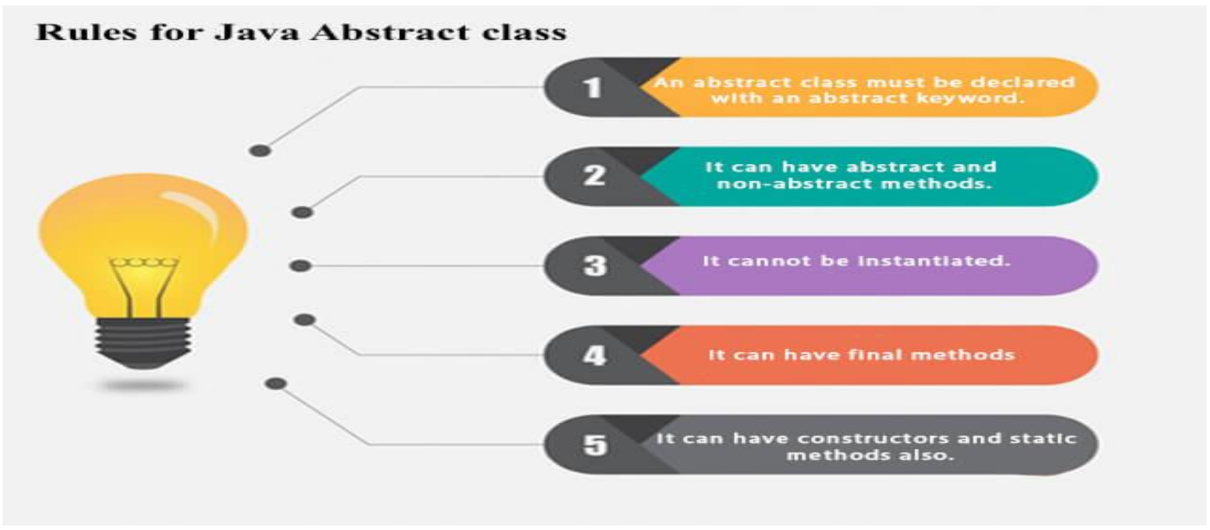
```
Animal myObj = new Animal(); // will generate an error
```

to access the abstract class, it must be inherited from another class. Let's convert the Animal class we used in the Polymorphism chapter to an abstract class:

Example

Get your own Java Server

```
// Abstract class
abstract class Animal {
  // Abstract method (does not have a body)
  public abstract void animalSound();
  // Regular method
  public void sleep() {
    System.out.println("Zzz");
  }
}

// Subclass (inherit from Animal)
class Pig extends Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The pig says: wee wee");
  }
}

class Main {
  public static void main(String[] args) {
    Pig myPig = new Pig(); // Create a Pig object
    myPig.animalSound();
    myPig.sleep();
}
}
```


Rules for Java Abstract class
1
An abstract class must be declared with an abstract keyword.
2
It can have abstract and non-abstract methods.
3
It cannot be instantiated.
4
It can have final methods
5
It can have constructors and static methods also.

Java Interfaces and their advantages

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.

Method Overriding in Java

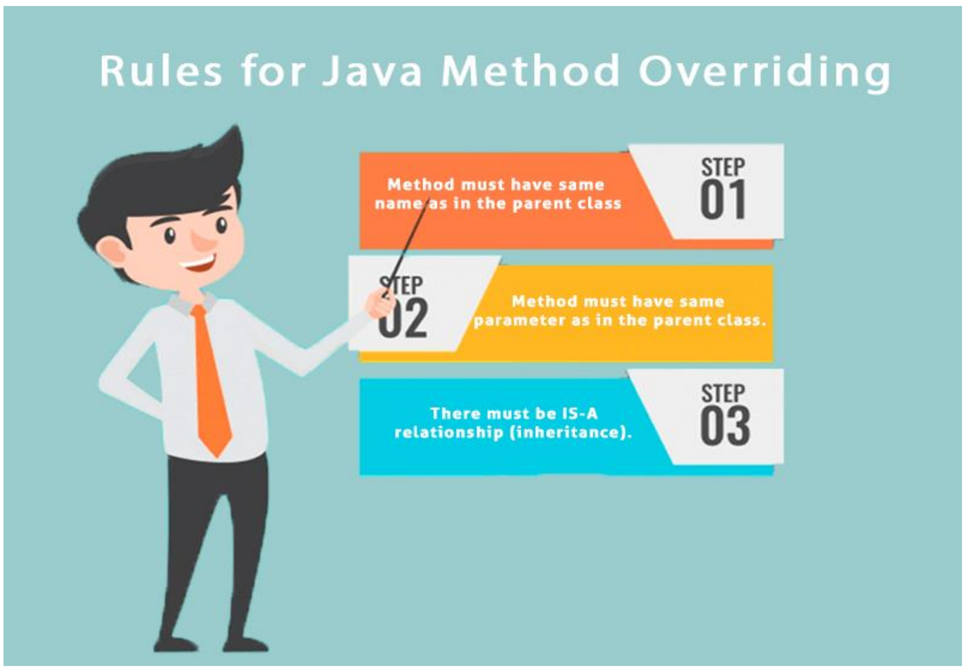
If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.

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.

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

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance).
4. 
Rules for Java Method Overriding
Method must have same name as in the parent class
STEP 01
STEP 02
Method must have same parameter as in the parent class.
There must be IS-A relationship (inheritance).
STEP 03

Example of method overriding

In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method are the same, and there is IS-A relationship between the classes, so there is method overriding.

```java
//Java Program to illustrate the use of Java Method Overriding
//Creating a parent class.
class Vehicle{
  //defining a method
  void run(){System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike2 extends Vehicle{
  //defining the same method as in the parent class
  void run(){System.out.println("Bike is running safely");}

  public static void main(String args[]){
  Bike2 obj = new Bike2();//creating object
  obj.run();//calling method
  }
}
```

Output:

```
Bike is running safely
```

Polymorphism in Java

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

Like we specified in the previous chapter; Inheritance lets us inherit attributes and methods from another class. Polymorphism uses those methods to perform different tasks. This allows us to perform a single action in different ways.

For example, think of a superclass called Animal that has a method called animalSound(). Subclasses of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.):

Example

Get your own Java Server

```
class Animal {
  public void animalSound() {
    System.out.println("The animal makes a sound");
  }
}

class Pig extends Animal {
  public void animalSound() {
    System.out.println("The pig says: wee wee");
  }
}

class Dog extends Animal {
  public void animalSound() {
    System.out.println("The dog says: bow wow");
  }

}
```

Creating, Implementing And Extending Interfaces

An interface in Java is syntactically similar to a class but can have only abstract methods declaration and constants as members.
In other words, an interface is a collection of abstract methods and constants (i.e. static and final fields). It is used to achieve complete abstraction.

Every interface in Java is abstract by default. So, it is not compulsory to write abstract keyword with an interface. Once an interface is defined, we can create any number of separate classes and can provide their own implementation for all the abstract methods defined by an interface.

A class that implements an interface is called implementation class. A class can implement any number of interfaces in Java. Every implementation class can have its own implementation for abstract methods specified in the interface as shown in the below figure

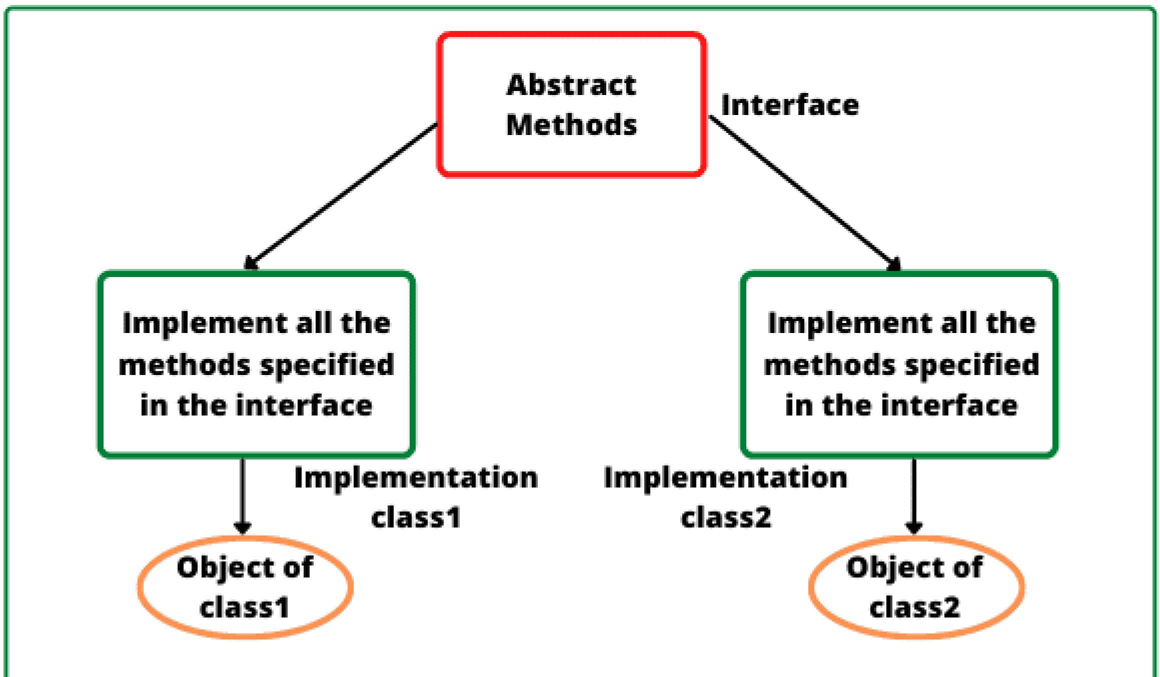

Abstract Methods
Interface
Implement all the methods specified in the interface
Implement all the methods specified in the interface
Implementation class1
Implementation class2
Object of class1
Object of class2

Fig: Interface and its Implementation Classes

Why do We Use Interface?

There are mainly five reasons or purposes of using an interface in Java. They are as follows:

1. In industry, architect-level people create interfaces, and then they are given to developers for writing classes by implementing interfaces provided.

2. Using interfaces is the best way to expose our project's API to some other projects. In other words, we can provide interface methods to the third-party vendors for their implementation.

For example, HDFC bank can expose methods or interfaces to various shopping carts.

3. Programmers use interface to customize features of software differently for different objects.

4. It is used to achieve full abstraction in java.

5. By using interfaces, we can achieve the functionality of multiple inheritance.

How to Declare Interface in Java?

In Java, an interface is declared syntactically much like a class. It is declared by using the keyword interface followed by interface name. It has the following general form:

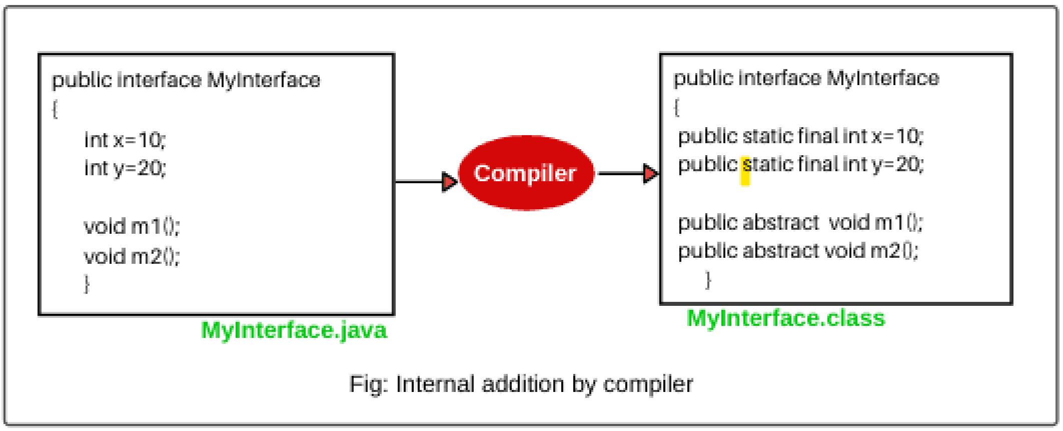
```
Syntax:
  accessModifier interface interfaceName
  {
    // declare constant fields.
    // declare methods that abstract by default.
  }
```

Before interface keyword, we can specify access modifiers such as public, or default with abstract keyword. Let's understand the declaration of an interface with the help of an example.

```
public abstract interface MyInterfac
{
  int x = 10; // public static final keyword invisibly present.
  void m1(); // public and abstract keywords invisibly present.
  void m2();
}
```

As you can see in the above example, both methods m1() and m2() defined in interface are declared with no body and do not have public or abstract modifiers present. The variable x declared in MyInterface is like a simple variable.

Java compiler automatically adds public and abstract keywords before to all interface methods. Moreover, it also adds public, static, and final keywords before interface variables. Look at the below figure to understand better.


```
public interface MyInterface
{
    int x=10;
    int y=20;

    void m1();
    void m2();
    }
```

```
public interface MyInterface
{
 public static final int x=10;
 public static final int y=20;

 public abstract  void m1();
 public abstract void m2();
    }
```

Fig: Internal addition by compiler

Note:

a) Earlier to Java 8, an interface could not define any implementation whatsoever. An interface can only declare abstract methods.

b) Java 8 changed this rule. From Java 8 onwards, it is also possible to add a default implementation to an interface method.

c) To support lambda functions, Java 8 has added a new feature to interface. We can also declare default methods and static methods inside interfaces.

d) From Java 9 onwards, an interface can also declare private methods.

Features of Interface

There are the following features of an interface in Java. They are as follows:

1. Interface provides pure abstraction in Java. It also represents the Is-A relationship.

2. It can contain three types of methods: abstract, default, and static methods.

3. All the (non-default) methods declared in the interface are by default abstract and public. So, there is no need to write abstract or public modifiers before them.

4. The fields (data members) declared in an interface are by default public, static, and final. Therefore, they are just public constants. So, we cannot change their value by implementing class once they are initialized.

5. Interface cannot have constructors.

Extending Interface in Java with Example

Like classes, an interface can also extend another interface. This means that an interface can be sub interfaces from other interfaces.

The new sub-interface will inherit all members of the super interface similar to subclasses. It can be done by using the keyword “extends”. It has the following general form:

Syntax:

```
 interface interfaceName2 extends interfaceName1
 {
   // body of interfaceName2.
 }
```

Look at the below figure a, b, c, and d to understand better.

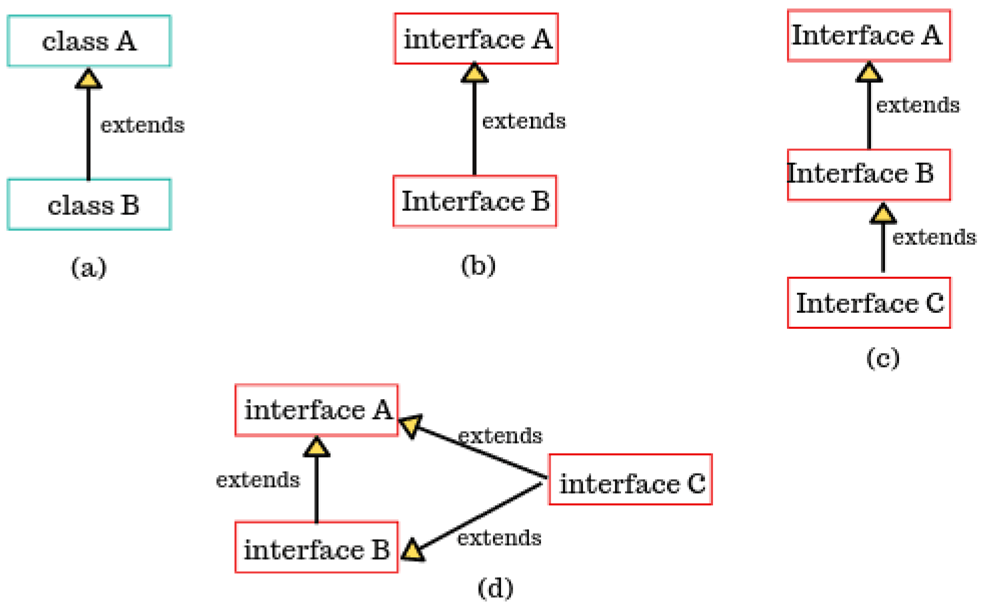


Fig: Various forms of extending Interface in Java

1. We can define all the constants in one interface and methods in another interface. We can use constants in classes where methods are not required. Look at the example below.

```
interface A
{
  int x = 10;
  int y = 20;
}
interface B extends A
{
  void show();
}
```

The interface B would inherit both constants x and y into it.

2. We can also extend various interfaces together by a single interface. The general declaration is given below:

```
interface A
{
  int x = 20;
  int y = 30;
}
```

```
interface B extends A
{
  void show();
 }
interface C extends A, B
{
  . . . . . . . .
 }
```

Key points:

1. An interface cannot extend classes because it would violate rules that an interface can have only abstract methods and constants.
2. An interface can extend Interface1, Interface2.

Implementing Interface in Java with Example

An interface is used as “superclass” whose properties are inherited by a class. A class can implement one or more than one interface by using a keyword implements followed by a list of interfaces separated by commas.

When a class implements an interface, it must provide an implementation of all methods declared in the interface and all its super interfaces.

Otherwise, the class must be declared abstract. The general syntax of a class that implements an interface is as follows:

```
Syntax:
1. accessModifier class className implements interfaceName
   {
    // method implementations;
    // member declaration of class;
   }
2. A more general form of interface implementation is given below:
  accessModifier class className extends superClass implements interface1, interface2,.. .
  {
     // body of className.
```

}

This general form shows that a class can extend another class while implementing interfaces.

Key points:

1. All methods of interfaces when implementing in a class must be declared as public otherwise, you will get a compile-time error if any other modifier is specified.
2. Class extends class implements interface.
3. Class extends class implements Interface1, Interface2…

The implementation of interfaces can have the following general forms as shown in the below figure.

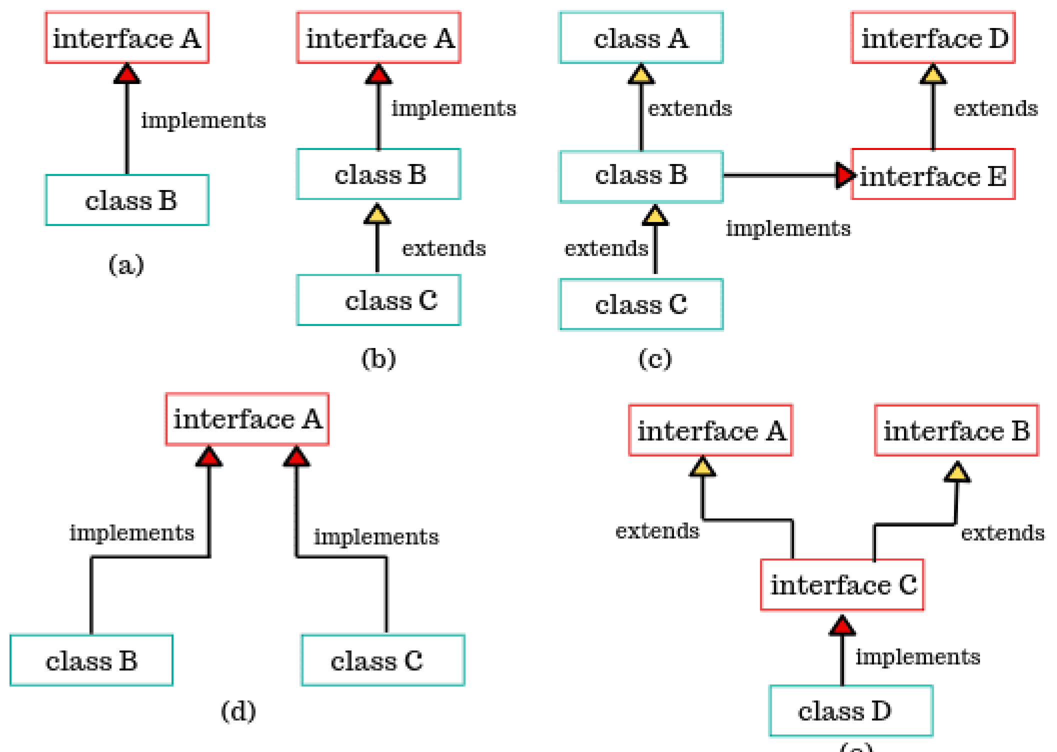


Fig: Various forms of interface implementation

Packages in JAVACreating and Using

In this tutorial, we are going to discuss **packages in Java** with the help of example programs. In small projects, all the Java files have unique names. So, it is not difficult to put them in a single folder.

But, in the case of huge projects where the number of Java files is large, it is very difficult to put files in a single folder because the manner of storing files would be disorganized.

Moreover, if different Java files in various modules of the project have the same name, it is not possible to store two Java files with the same name in the same folder because it may occur naming conflict.

This problem of naming conflict can be overcome by using the concept of packages. In Java, APIs consist of one or more packages where packages consist of many classes, classes contain several methods and fields.

When you create an application in Java, you should create a proper folder structure for better reusability, maintenance, and avoiding naming conflict but How?

What is Package in Java

A package is nothing but a physical folder structure (directory) that contains a group of related classes, interfaces, and sub-packages according to their functionality. It provides a convenient way to organize your work. The Java language has various in-built packages.

For example, java.lang, java.util, java.io, and java.net. All these packages are defined as a very clear and systematic packaging mechanism for categorizing and managing.

Let’s understand it with the help of real-time examples.

Realtime Example of Packages in Java

A real-life example is when you download a movie, song, or game, you make a different folder for each category like movie, song, etc. In the same way, a group of packages in java is just like a library.

The classes and interfaces of a package are like books in the library that can reuse several times when we need them. This reusability nature of packages makes programming easy.

Therefore, when you create any software or application in Java programming language, they contain hundreds or thousands of individual classes and interfaces.

Advantage of using Packages in Java

1. **Maintenance:** Java packages are used for proper maintenance. If any developer newly joined a company, he can easily reach to files needed. 2. **Reusability:** We can place the common code in a common folder so that everybody can check that folder and use it whenever needed.

3. **Name conflict:** Packages help to resolve the naming conflict between the two classes with the same name. Assume that there are two classes with the same name Student.java. Each class will be stored in its own packages such as stdPack1 and stdPack2 without having any conflict of names.

4. **Organized:** It also helps in organizing the files within our project.

5. **Access Protection:** A package provides access protection. It can be used to provide visibility control. The members of the class can be defined in such a manner that they will be visible only to elements of that package.

Types of Packages in Java

There are mainly two types of packages available in Java. They are:

- User-defined package
- built-in package (also called predefined package)

Let’s understand first user-defined package and how to create it easily in a Java program.

User-defined Package in Java

The package which is defined by the user is called user-defined or custom package in Java. It contains user-defined classes and interfaces. Let’s understand how to create a user-defined package.

Creating User-defined Package

Java supports a keyword called “package” which is used to create user-defined packages in Java programming. The general syntax to create a package is as:

```
package packageName;
```

Here, packageName is the name of package. The package statement must be the first line in a Java source code file followed by one or more classes. For example:

```
package myPackage;
public class A {
    // class body
}
```

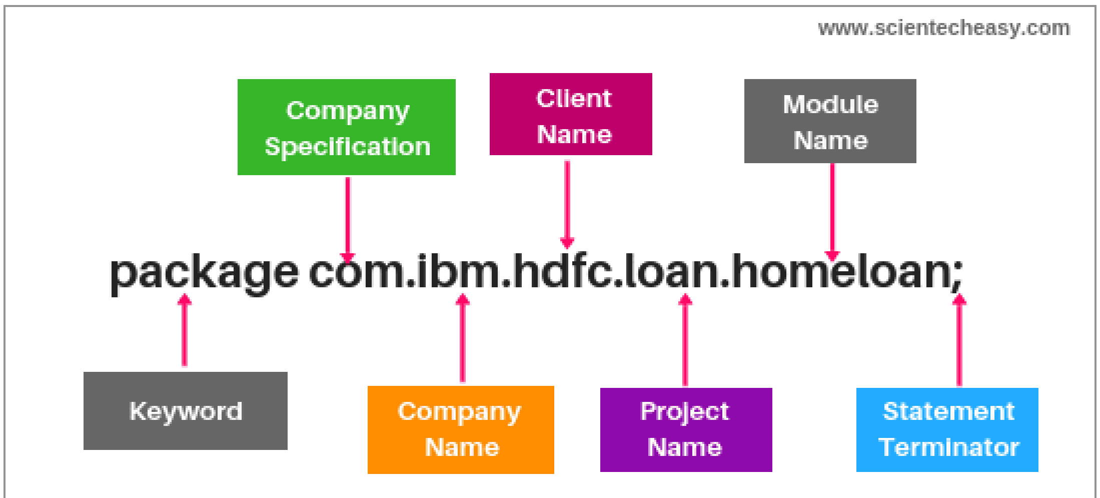
In this example, myPackage is the name of package. The statement “package myPackage;” signifies that the class “A” belongs to the “myPackage” package.

Naming Convention for User-defined Package in Realtime Project

---

While developing your project, you must follow some naming conventions regarding packages declaration. Let’s take an example to understand the convention.

Look at the below a complete package structure of the project.



```
package com.ibm.hdfc.loan.homeloan;
```

While declaring the package name, every character should be lowercase

Fig: Complete Package Structure of Project

Abstract Windowing Tool Kit

- Introduction to user interface and AWT components and containers
- Introduction to AWT UI controls, hierarchy and their features
- Introduction to event handling
- Introduction to event handling classes
- Introduction to event listener interfaces
- Introduction to AWT Layouts

Abstract Windowing Toolkit

The Abstract Windowing Toolkit (AWT) is a library of Java GUI object classes that is included with the Java Development Kit from Sun Microsystems. The AWT handles common interface elements for windowing environments including Windows.

The AWT contains the following set of GUI components:

Button

CheckBox

CheckBox Group (RadioList)

Choice (PopupList)

Label (StaticText)

List (ListBox)

Scroll Bar

Text Component (TextField)

Menu

Introduction to user interface and AWT components and containers

Introduction to the AWT:-

A description of Java's user interface toolkit

The Java programming language class library provides a user interface toolkit called the Abstract Windowing Toolkit, or the AWT. The AWT is both powerful and flexible. Newcomers, however, often find that its power is veiled. The class and method descriptions found in the distributed documentation provide little guidance for the new programmer. Furthermore, the available examples often leave many important questions unanswered. Of course, newcomers should expect some difficulty. Effective graphical user interfaces are inherently challenging to design and implement, and the sometimes complicated interactions between classes in the AWT only make this task more complex. However, with proper guidance, the creation of a graphical user interface using the AWT is not only possible, but relatively straightforward.

This article covers some of the philosophy behind the AWT and addresses the practical concern of how to create a simple user interface for an applet or application.

What is a user interface:-

The user interface is that part of a program that interacts with the user of the program. User interfaces take many forms. These forms range in complexity from simple command-line interfaces to the point-and-click graphical user interfaces provided by many modern applications.

[Also on InfoWorld: Why software engineering estimates are garbage]

At the lowest level, the operating system transmits information from the mouse and keyboard to the program as input, and provides pixels for program output. The AWT was designed so that programmers don't have worry about the details of tracking the mouse or reading the keyboard, nor attend to the details of writing to the screen. The AWT provides a well-designed object-oriented interface to these low-level services and resources.

Because the Java programming language is platform-independent, the AWT must also be platform-independent. The AWT was designed to provide a common set of tools for graphical user interface design that work on a variety of platforms. The user interface elements provided by the AWT are implemented using each platform's native GUI toolkit, thereby preserving the look and feel of each platform. This is one of the AWT's strongest points. The disadvantage of such an approach is the fact that a graphical user interface designed on one platform may look different when displayed on another platform.

Components and containers:-

A graphical user interface is built of graphical elements called components. Typical components include such items as buttons, scrollbars, and text fields. Components allow the user to interact with the program and provide the user with visual feedback about the state of the program. In the AWT, all user interface components are instances of class Component or one of its subtypes.

Components do not stand alone, but rather are found within containers. Containers contain and control the layout of components. Containers are themselves components, and can thus be placed inside other containers. In the AWT, all containers are instances of class Container or one of its subtypes.

Spatially, components must fit completely within the container that contains them. This nesting of components (including containers) into containers creates a tree of elements, starting with the container at the root of the tree and expanding out to the leaves, which are components such as buttons.

Types of containers:-

The AWT provides four container classes. They are class Window and its two subtypes -- class Frame and class Dialog -- as well as the Panel class. In addition to the containers provided by the AWT, the Applet class is a container -- it is a subtype of the Panel class and can therefore hold components. Brief descriptions of each container class provided by the AWT are provided below.

Window A top-level display surface (a window). An instance of the Window class is not attached to nor embedded within another container. An instance of the Window class has no border and no title.

Frame A top-level display surface (a window) with a border and title. An instance of the Frame class may have a menu bar. It is otherwise very much like an instance of the Window class.

Dialog A top-level display surface (a window) with a border and title. An instance of the Dialog class cannot exist without an associated instance of the Frame class.

Panel

A generic container for holding components. An instance of the Panel class provides a container to which to add components.

Creating a container

Before adding the components that make up a user interface, the programmer must create a container. When building an application, the programmer must first create an instance of class Window or class Frame. When building an applet, a frame (the browser window) already exists. Since the Applet class is a subtype of the Panel class, the programmer can add the components to the instance of the Applet class itself.

The code in Listing 1 creates an empty frame. The title of the frame ("Example 1") is set in the call to the constructor. A frame is initially invisible and must be made visible by invoking its show() method.

```
import java.awt.*;

public class Example1

{

  public static void main(String [] args)

  {

          Frame f = new Frame("Example 1");

          f.show();

  }

}
```

Adding components to a container

To be useful, a user interface must consist of more than just a container -- it must contain components. Components are added to containers via a container's add() method. There are three basic forms of the add() method. The method to use depends on the container's layout manager (see the section titled Component layout).

The code in Listing 4 adds the creation of two buttons to the code presented in Listing 3. The creation is performed in the init() method because it is automatically called during applet initialization. Therefore, no matter how the program is started, the buttons are created, because init() is called by either the browser or by the main() method. Figure 6 contains the resulting applet.

```
import java.awt.*;

public class Example3 extends java.applet.Applet

{

  public void init()

  {

        add(new Button("One"));

        add(new Button("Two"));

  }

  public Dimension preferredSize()

  {

        return new Dimension(200, 100);

  }

  public static void main(String [] args)

  {

        Frame f = new Frame("Example 3");

        Example3 ex = new Example3();

        ex.init();

        f.add("Center", ex);

        f.pack();

        f.show();

  }
}
```

Java AWT Hierarchy:-

The hierarchy of Java AWT classes are given below.

Introduction to event Handling

What is an Event?

Change in the state of an object is known as event i.e. event describes the change in state of source. Events are generated as result of user interaction with the graphical user interface components. 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.

Types of Event

The events can be broadly classified into two categories:

Foreground Events - Those events which require the direct interaction of user.They are generated as consequences of a person interacting with the graphical components in 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 require the interaction of end user are known as background events. Operating system interrupts, hardware or software failure, timer expires, an operation completion are the example of background events.

What is Event Handling?

Event Handling is the mechanism that controls the event and decides what should happen if an event occurs. This mechanism have the code which is known as event handler that is executed when an event occurs. Java Uses the Delegation Event Model to handle the events. This model defines the standard mechanism to generate and handle the events.Let's have a brief introduction to this model.

The Delegation Event Model has the following key participants namely:

Source - The source is an object on which event occurs. Source is responsible for providing information of the occurred event to it's handler. Java provide as with classes for source object.

Listener - It is also known as event handler.Listener is responsible for generating response to an event. From java implementation point of view the listener is also an object. Listener waits until it receives an event. Once the event is received , the listener process the event an then returns.

The benefit of this approach is that the user interface logic is completely separated from the logic that generates the event. The user interface element is able to delegate the processing of an event to the separate piece of code. In this model ,Listener needs to be registered with the source object so that the listener can receive the event notification. This is an efficient way of handling the event because the event notifications are sent only to those listener that want to receive them.

Steps involved in event handling

The User clicks the button and the event is generated.

Now the object of concerned event class is created automatically and information about the source and the event get populated with in same object.

Event object is forwarded to the method of registered listener class.

the method is now get executed and returns.

Points to remember about listener:-

In order to design a listener class we have to develop some listener interfaces.These Listener interfaces forecast some public abstract callback methods which must be implemented by the listener class.

If you do not implement the any if the predefined interfaces then your class can not act as a listener class for a source object.

Callback Methods:-

These are the methods that are provided by API provider and are defined by the application programmer and invoked by the application developer. Here the callback methods represents an event method. In response to an event java jre will fire callback method. All such callback methods are provided in listener interfaces.

If a component wants some listener will listen to it's events the the source must register itself to the listener.

Event Handling Example:-

Create the following java program using any editor of your choice in say D:/ > AWT > com > tutorialspoint > gui >

AwtControlDemo.java

```
package com.tutorialspoint.gui;


import java.awt.*;

import java.awt.event.*;


public class AwtControlDemo {


   private Frame mainFrame;

   private Label headerLabel;

   private Label statusLabel;

   private Panel controlPanel;


   public AwtControlDemo(){

         prepareGUI();

   }


   public static void main(String[] args){

         AwtControlDemo  awtControlDemo = new AwtControlDemo();

         awtControlDemo.showEventDemo();
```

```
}


private void prepareGUI(){
        mainFrame = new Frame("Java AWT Examples");
        mainFrame.setSize(400,400);
        mainFrame.setLayout(new GridLayout(3, 1));
        mainFrame.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent windowEvent){
        System.exit(0);
        }
        });
        headerLabel = new Label();
        headerLabel.setAlignment(Label.CENTER);
        statusLabel = new Label();
        statusLabel.setAlignment(Label.CENTER);
        statusLabel.setSize(350,100);

        controlPanel = new Panel();
        controlPanel.setLayout(new FlowLayout());

        mainFrame.add(headerLabel);
```

```
      mainFrame.add(controlPanel);

      mainFrame.add(statusLabel);

      mainFrame.setVisible(true);

}



private void showEventDemo(){

      headerLabel.setText("Control in action: Button");



      Button okButton = new Button("OK");

      Button submitButton = new Button("Submit");

      Button cancelButton = new Button("Cancel");



  okButton.setActionCommand("OK");

  submitButton.setActionCommand("Submit");

  cancelButton.setActionCommand("Cancel");



      okButton.addActionListener(new ButtonClickListener());

      submitButton.addActionListener(new ButtonClickListener());

      cancelButton.addActionListener(new ButtonClickListener());



      controlPanel.add(okButton);
```

```
}

          controlPanel.add(submitButton);

          controlPanel.add(cancelButton);



          mainFrame.setVisible(true);

   }



   private class ButtonClickListener implements ActionListener{

          public void actionPerformed(ActionEvent e) {

          String command = e.getActionCommand();

          if( command.equals( "OK" ))  {

          statusLabel.setText("Ok Button clicked.");

          }

          else if( command.equals( "Submit" ) )  {

          statusLabel.setText("Submit Button clicked.");

          }

          else  {

                 statusLabel.setText("Cancel Button clicked.");

          }

          }

   }

}
```

Event Handling Classes

EventObject class:-

It is the root class from which all event state objects shall be derived. All Events are constructed with a reference to the object, the source, that is logically deemed to be the object upon which the Event in question initially occurred upon.This class is defined in java.util package.

Class declaration

Following is the declaration for java.util.EventObject class:

public class EventObject extends Object implements Serializable Field Following are the fields for java.util.EventObject class:

protected Object source -- The object on which the Event initially occurred.

Class constructors

S.N. Constructor & Description

1

EventObject(Object source)

Constructs a prototypical Event.

Class methods

S.N. Method & Description

1 Object getSource()

The object on which the Event initially occurred.

2 String toString()

Returns a String representation of this EventObject.

Methods inherited

This class inherits methods from the following classes:

java.lang.Object

AWT Event Classes:

Following is the list of commonly used event classes.

Sr. No. **Control & Description**

1 AWTEvent

It is the root event class for all AWT events. This class and its subclasses supercede the original java.awt.Event class.

2 ActionEvent

The ActionEvent is generated when button is clicked or the item of a list is double clicked.

3 InputEvent

The InputEvent class is root event class for all component-level input events.

4 KeyEvent

On entering the character the Key event is generated.

5 MouseEvent

This event indicates a mouse action occurred in a component.

Event Listner interface

The Event listener represent the interfaces responsible to handle events. Java provides us various Event listener classes but we will discuss those which are more frequently used. Every method of an event listener method has a single argument as an object which is subclass of EventObject class. For example, mouse event listener methods will accept instance of MouseEvent, where MouseEvent derives from EventObject.

Event Listner interface:-

It is a marker interface which every listener interface has to extend.This class is defined in java.util package.

Class declaration:-

Following is the declaration for java.util.EventListener interface:

public interface EventListener

AWT Event Listener Interfaces:

Following is the list of commonly used event listeners.

<u>Sr. No.</u> **<u>Control & Description</u>**

1 ActionListener

This interface is used for receiving the action events.

2 ComponentListener

This interface is used for receiving the component events.

3 ItemListener

This interface is used for receiving the item events.

4 KeyListener

This interface is used for receiving the key events.

5 MouseListener

This interface is used for receiving the mouse events.

6 TextListener

This interface is used for receiving the text events.

7 WindowListener

This interface is used for receiving the window events.

AWT Layouts

Introduction

Layout means the arrangement of components within the container. In other way we can say that placing the components at a particular position within the container. The task of layouting the controls is done automatically by the Layout Manager.

Layout Manager:-

The layout manager automatically positions all the components within the container. If we do not use layout manager then also the components are positioned by the default layout manager. It is possible to layout the controls by hand but it becomes very difficult because of the following two reasons.

1. It is very tedious to handle a large number of controls within the container.
2. Oftenly the width and height information of a component is not given when we need to arrange them

Java provide us with various layout manager to position the controls. The properties like size,shape and arrangement varies from one layout manager to other layout manager. When the size of the applet or the application window changes the size, shape and arrangement of the components also changes in response i.e. the layout managers adapt to the dimensions of appletviewer or the application window.

The layout manager is associated with every Container object. Each layout manager is an object of the class that implements the LayoutManager interface.

Following are the interfaces defining functionalities of Layout Managers.

__Sr. No.__ __Interface & Description__

(1) LayoutManager

The LayoutManager interface declares those methods which need to be implemented by the class whose object will act as a layout manager.

(2) LayoutManager2

The LayoutManager2 is the sub-interface of the LayoutManager.This interface is for those classes that know how to layout containers based on layout constraint object.

AWT Layout Manager Classes:

Following is the list of commonly used controls while designed GUI using AWT.

__Sr. No__. __LayoutManager & Description__

(1) BorderLayout

The borderlayout arranges the components to fit in the five regions: east, west, north, south and center.

(2) CardLayout

The CardLayout object treats each component in the container as a card. Only one card is visible at a time.

(3) FlowLayout

The FlowLayout is the default layout.It layouts the components in a directional flow.

(4) GridLayout

The GridLayout manages the components in form of a rectangular grid.

(5) GridBagLayout

This is the most flexible layout manager class.The object of GridBagLayout aligns the component vertically,horizontally or along their baseline without requiring the components of same size.