Automation Testing

Introduction to Core Java

Learning Objectives

By the end of this lesson, you will be able to:

- Understand Core Java and its features
- Implement all the OOPs concepts
- Understand the concepts of strings, arrays, and collections
- Perform multithreading, file handling, and exception handling in Java

Java: Overview

Introduction to Java

Java is a programming language and computing platform for developing application software that can run on any device.

It is used in a wide variety of platforms, from game consoles, laptops, and cell phones to data centers, enterprise servers, and scientific supercomputers.

Why Java?

These are the reasons why programmers should use Java:

- Simple and easy-to-read
- Works on object-oriented programming (OOPs) concepts, which makes it robust and secure.
- Platform-independent as it has its own runtime environment called JRE (Java Runtime Environment)
- Provides a strong networking infrastructure for server-side computing
- Comprises well-defined libraries modified from the existing C libraries

Features of Java

Features of Java

These are some features that make Java easy to use:

- Object-oriented
- Platform-independent
- Architecture-neutral and portable
- Interpretable

Fundamental Components of Java

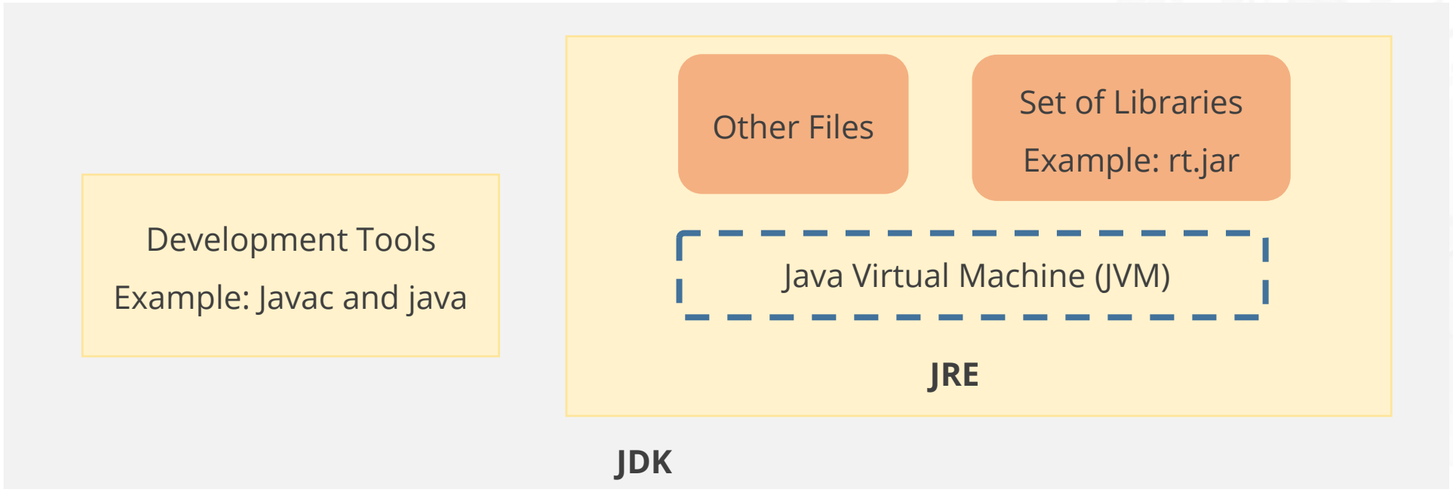
Fundamental Components of Java

Java Development Kit (JDK)

Java Runtime Environment (JRE)

Java Virtual Machine (JVM)

JDK contains JRE and development tools necessary to compile, document, and package Java programs.



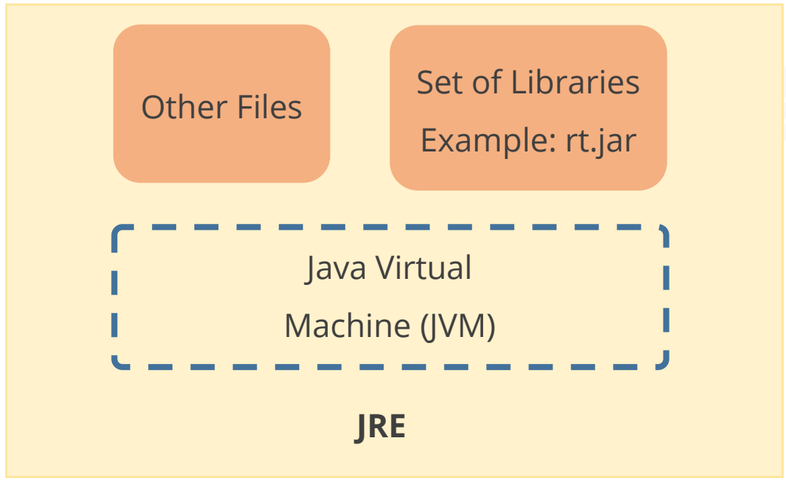
Fundamental Components of Java

Java Development Kit (JDK)

Java Runtime Environment (JRE)

Java Virtual Machine (JVM)

JRE is a runtime environment in which Java bytecode is executed. It contains all the class libraries and other files that JVM uses at runtime.



Fundamental Components of Java

Java Development Kit (JDK)

Java Runtime Environment (JRE)

Java Virtual Machine (JVM)

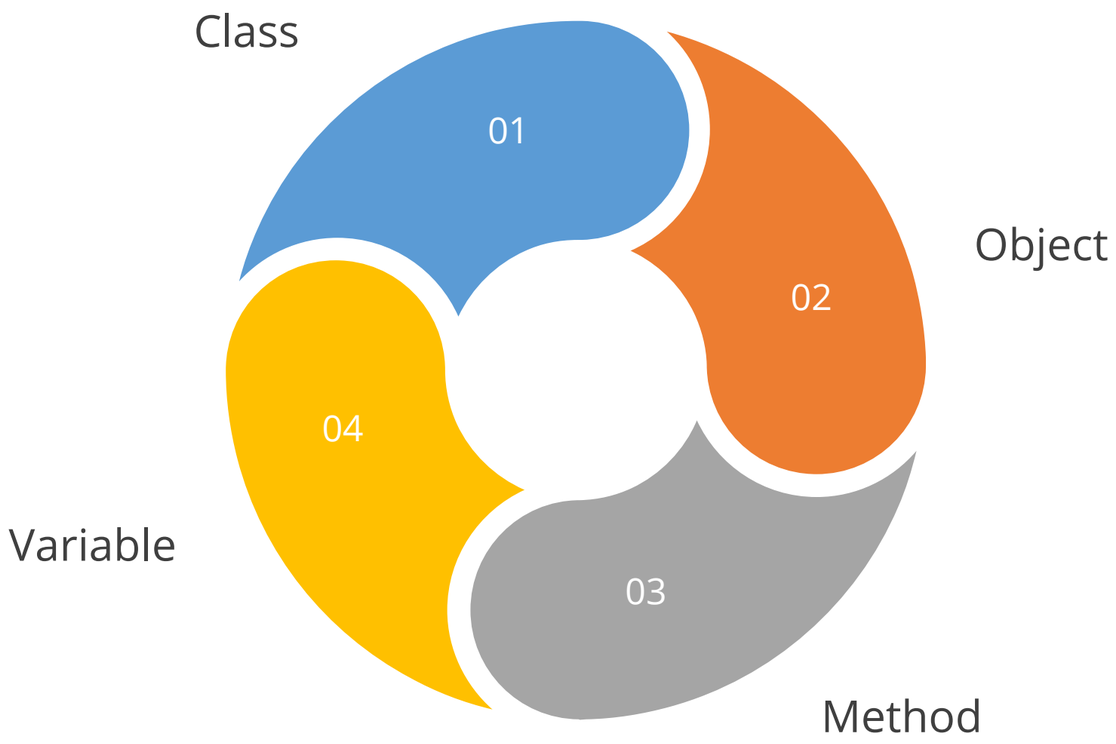
JVM is an abstract machine. It is a specification that provides runtime environment in which Java bytecode can be executed. Unlike other languages, Java “executables” are executed on a CPU that does not exist.

Java Virtual Machine (JVM)

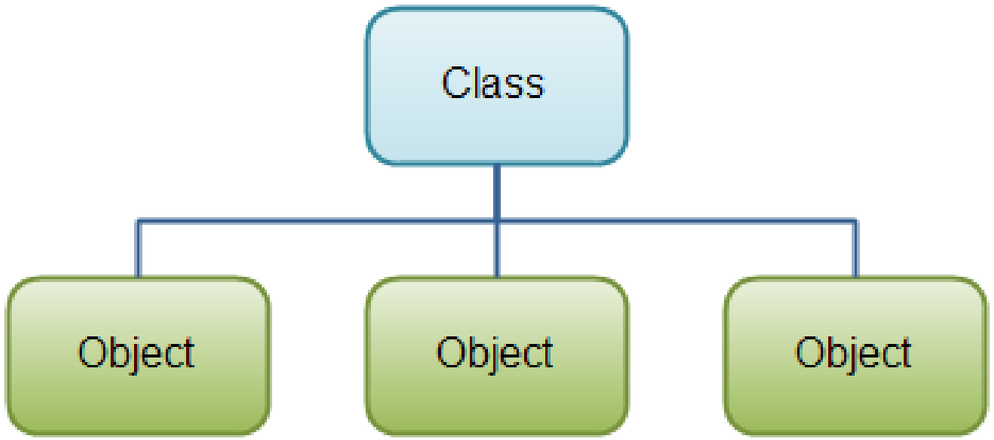
Core Elements

Core Elements

There are four core elements in Java, which include:



Core Elements : Class



A class is a blueprint or prototype for creating objects, providing initial values for the variable state, and implementing behavior.

Core Elements : Class

A class has two components:

- Class declaration: Class name, access modifier
- Class body: Methods, variables, constructors, objects, and inner classes.

Syntax for using **class**:

```
access_modifier class ClassName {

/*
class consists of
a)class level or global variables
b)objects
c) static and non static methods
d) constructors
e) inner classes
*/

}
```

Core Elements : Object

01 An object is a software bundle of related states and behavior.

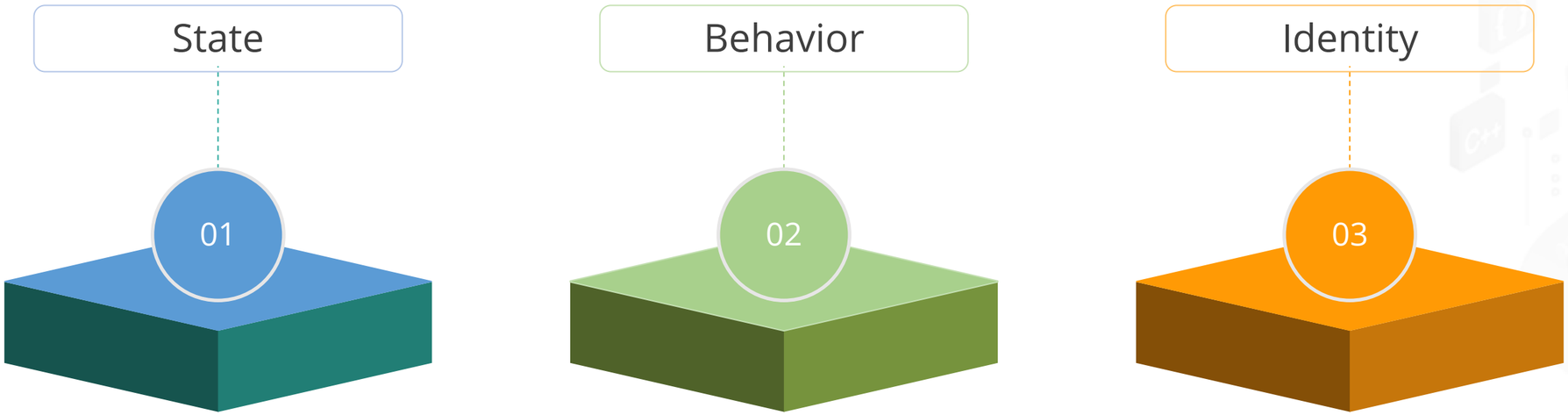
02 Software objects are used to model the real-world objects.

03 It is an entity that has “variable” and “method.”

04 Its syntax is
“ClassName objectName = **new** ClassName();”.

Core Elements : Object

An object has three characteristics:



Core Elements : Methods

A method is a collection of statements that are grouped to perform an operation.

Syntax of using **method**:

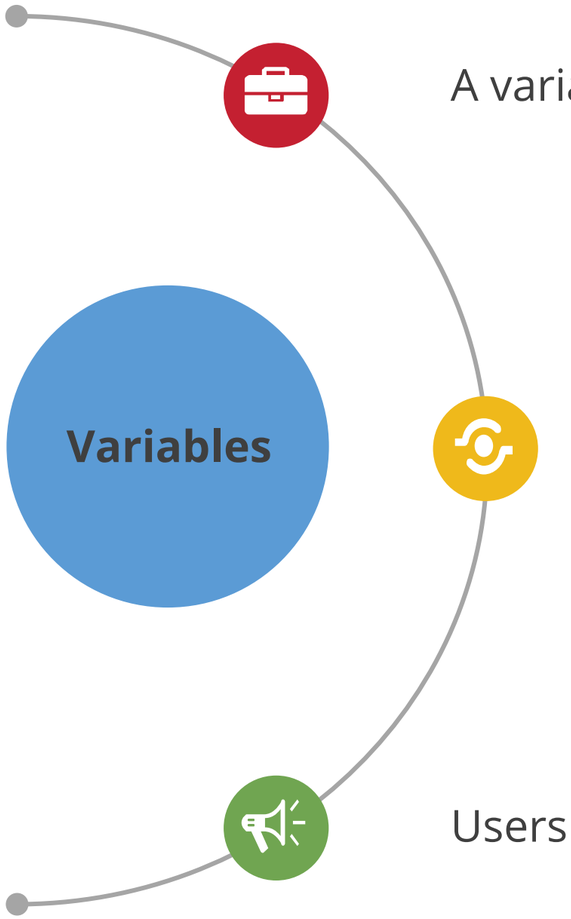
```
access_modifier return_type methodName (arguments list)
{

/*
code statements
*/

}
```

Core Elements : Methods

Example of a **method**:

```
public void findOddEvenNumber(int num)
{
//method body
if(num%2==0)
System.out.println(num+" is even");
else
System.out.println(num+" is odd");
}
```

Core Elements : Variables

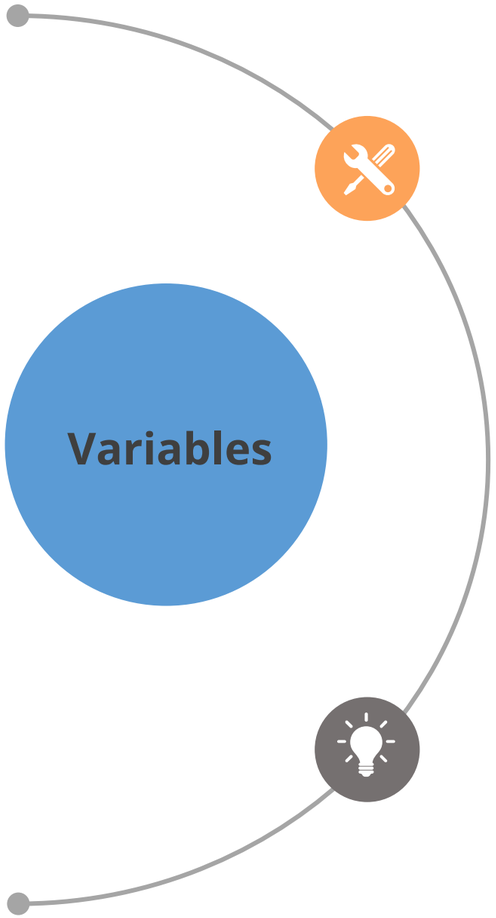


A variable is a named storage that a program uses.

Each variable has a specific type, which determines the value that can be stored within it and the operations that can be applied to it.

Users must declare the variable before using it.

Core Elements : Variables



A variable can be declared using the eight primitive data types: byte, short, int, long, float, double, char, or boolean.

Declaring a variable only reserves a name and some memory for it, whereas initializing the variable gives the variable a value.

Core Elements : Variables

Syntax for declaring and initializing a variable:

dataType variableName1 = value1, variableName2 = value2, ... ;

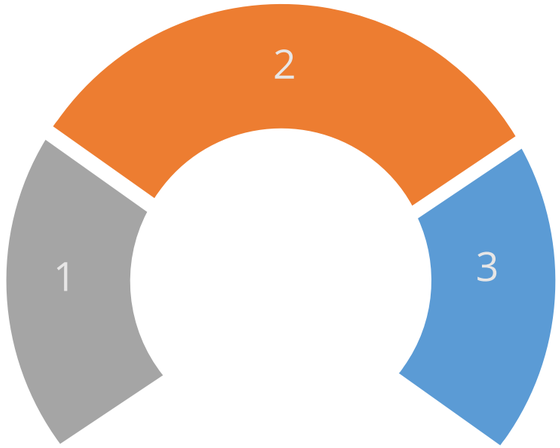
```
class A{
int a,b,c;          //declaration of three variables
int d=100, e=20, f;   //initialization of d and e and declaration of f
byte = 25;          //initialization, where a “byte” variable b is given a value of 25
}
}//end of class
```

Core Elements : Variables

There are three types of variables in Java, which include:

Instance variable: A non-static variable that is declared inside the class but outside the method

Local variable: A variable declared inside methods, constructors, or blocks



Class or Static variable: A variable that is declared as static and that cannot be local

Core Elements : Variables

Example of types of variables:

```
class A{
int data=50;            //instance variable
static int m=100;     //static variable
void method(){
int n=90;              //local variable
}
}//end of class
```

Keywords

Keywords

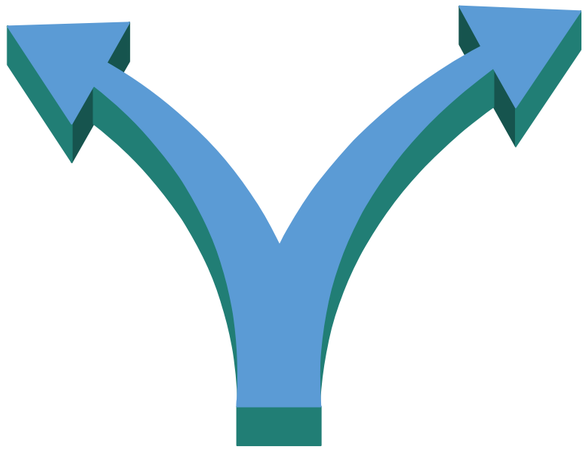
- In Java, keywords are a set of reserved words which cannot be used as a variable name, object name or class name.
- They act as a key to respective predefined code in the Java libraries.
- They are also referred to as reserved words in some cases.

Data Types

Data Types

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

Primitive:
Boolean, char, byte, short, int, long, float, and double



Non-Primitive:
Classes, Interfaces, and Arrays

Primitive Data Types

- byte
- short
- int
- long
- float
- double
- boolean
- char

- 8-bit integer
- Default value of 0
- Range: -128 and 127
- Example: **byte a=100**

Primitive Data Types

- byte
- short
- int
- long
- float
- double
- boolean
- char

- 16-bit integer
- Default value of 0
- Range: -32768 and 32767
- Example: **short s = 10000**

Primitive Data Types

- byte
- short
- **int**
- long
- float
- double
- boolean
- char

- 32-bit integer
- Default value of 0
- Ranges: -2147483648 and 2147483647
- Example: **int a = 100000**

Primitive Data Types

- byte
- short
- int
- long
- float
- double
- boolean
- char

- 64-bit integer
- Default value of 0L
- Range: -2^63 and 2^63 -1
- Example: **long a = 100000L**

Primitive Data Types

- byte
- short
- int
- long
- float
- double
- boolean
- char

- 32-bit floating point
- Default value of 0.0f
- Range: 6 and 7 decimals
- Example: **float f1 = 234.5f**

Primitive Data Types

- byte
- short
- int
- long
- float
- **double**
- boolean
- char

- 64-bit floating point
- Default value of 0.0d
- Range: 15 decimals
- Example: **double d1 = 123.4**

Primitive Data Types

- byte
- short
- int
- long
- float
- double
- **boolean**
- char

- Data type of one bit
- Default value of false
- Range: True or False
- Example: **boolean one = true**

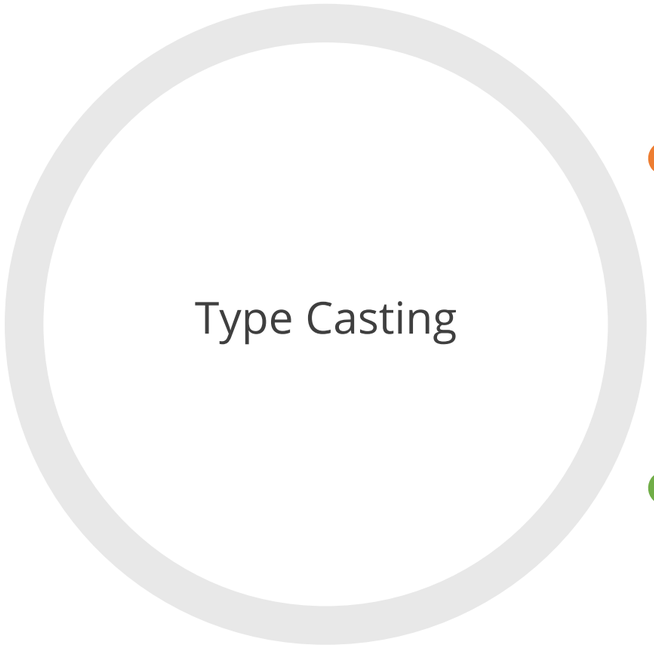
Primitive Data Types

- byte
- short
- int
- long
- float
- double
- boolean
- char

- Single 16-bit Unicode character
- Range: ‘\u0000’ (0) and \uffff (65535)
- Example: **char letterA = 'A'**

Type Casting

When two data types are compatible with each other, the value of one data type is assigned to the other, which is called type casting.



Implicit Type Casting:
byte -> short -> int -> long -> float -> double

Explicit Type Casting:
double -> float -> long -> int -> short -> byte

Type Casting

Implicit or Widening Conversion

- Widening or implicit conversion takes place when two data types are automatically converted.
- This is possible when:
  - The two data types are compatible
  - The value of a smaller data type is assigned to a bigger data type
- Conversion of numeric data type to char or boolean is not supported.

Explicit or Narrowing Conversion

- Users have to perform narrowing or explicit conversion to assign the value of a large data type to a smaller data type.
- This is useful for incompatible data types when automatic conversion is not possible.

Operators

Operators

Operators are special symbols that are used to perform specific operations, which return a result.

They are taken from other languages, and they behave as expected.

They can be categorized into the following groups:

- Arithmetic
- Unary
- Relational
- Shift
- Bitwise
- Logical
- Assignment
- Ternary

Operators

Arithmetic operators are used in mathematical expressions. They perform addition, subtraction, multiplication, and division.

Operator	Description
+	Additive operator (also used for String concatenation)
-	Subtraction operator
*	Multiplication operator
/	Division operator
%	Remainder operator
++	Increment operator
--	Decrement operator

Operators

Unary operators are used with only one operand.

Operator	Description
++	Increment operator
--	Decrement operator
!	Boolean value inverter

They perform increment and decrement of a value and inversion of a boolean value.

Operators

Relational operators determine if one operand is greater than, less than, equal to, or not equal to another operand.

Operator	Description
==	Equal to
!=	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

Operators

Shift operators shift the bits of a number either left or right. They can be used while multiplying or dividing a number by two.

Operator	Description
<<	Signed left shift
>>	Signed right shift
>>>	Unsigned right shift

Operators

Bitwise operators perform bitwise and bit shift operations on integral types, long, int, short, char, and byte.

Operator	Description
~	Unary bitwise complement
&	Bitwise AND
^	Bitwise exclusive OR
|	Bitwise inclusive OR

Operators

Logical operators are used to check multiple conditions together.

Operator	Description
&&	Conditional-AND
||	Conditional-OR
?:	Ternary (shorthand for if-then-else statement)

They return a Boolean result: the value 0 is not interpreted automatically as false and non-zero values are not interpreted automatically as true.

Operators

Assignment operators are used to assign value to a variable. Their associativity is from right to left.

Operator	Description
=	Assigns value on the right side to the variable on the left side
+=	Adds left operand with right operand and assigns the value to the left operand (variable)
-=	Subtracts right operand from left operand and assigns the value to the left operand (variable)
*=	Multiplies left operand with the right operand and assigns the value to the left operand (variable)
/=	Divides right operand from left operand and assigns the value to the left operand (variable)
%=	Calculates modulo of left operand by right operand and assigns the value to the left operand (variable)

Operators

Ternary operator is very commonly used in Java programming for replacing the if-then-else statement with one line.

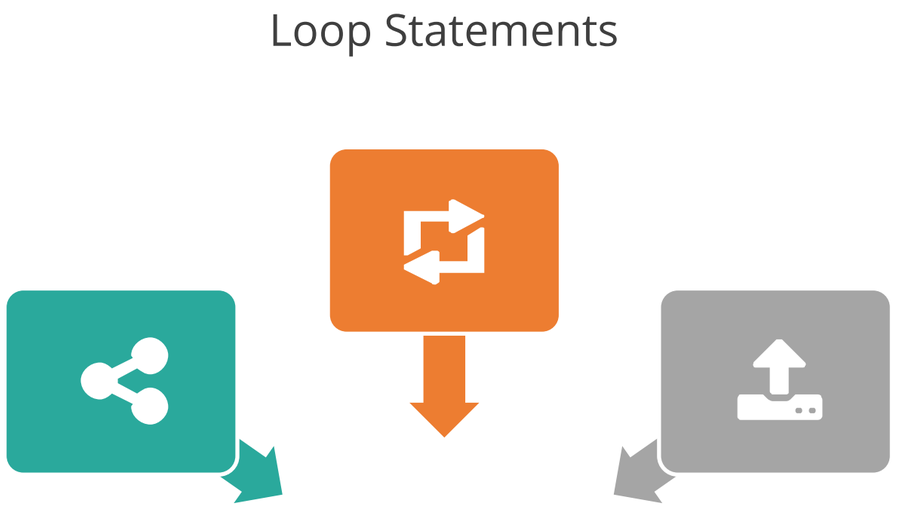
Operator	Description
? :	Condition ? true : false
||	Conditional-OR
?:	Ternary (shorthand for if-then-else statement)

It can only be used with three operands and hence the name ternary.

Control Statements

Control Statements

Java provides control statements which can change or control the flow of a program as per the requirement. That eventually provides smooth flow for the program.



Decision-Making Statements

Decision-Making Statements

if statement evaluates a boolean expression and lets the program to execute a block of code if the expression is true.

If the **if** statement is false and a separate block needs to be executed for the same, **else** statement can be used for it.

Decision-Making Statements

else-if statements are used if we have multiple statements to check after **if** statement is confirmed false.

In case we have a chain of statements, program may enter in the code block where the condition is true.

Decision-Making Statements

Syntax for using **if-else-if statements**:

```
if(condition 1) {
statement 1; //executes in case condition 1 is true
}
else if(condition 2) {
statement 2; //executes  in case condition 2 is true
}
else if(condition 3) {
statement 3; //executes  in case condition 3 is true
}
.
.
.
else {
statement 4; //executes  in case all the conditions are false
}
```

Decision-Making Statements

Example of **if-else-if** statements:

```
public class Student{

public static void main (String[] args) {

int rank = 20;

if(rank<10) {
System.out.println("You are in 1st section");
}
else if(rank>=10 && rank<20) {
System.out.println("You are in 2nd section");
}
else if(rank>=20 && rank<30 ) {
System.out.println("You are in 3rd section");
}
else {
System.out.println("You have failed");
}
}
}
```

Decision-Making Statements

if-else-if statements can also be used in nested format.

Nested **if** statements have **if** or **if-else** statements inside of another **if** or **if-else** statement.

Decision-Making Statements

Syntax for using nested **if-else-if statements**:

```
if(condition 1) {
if(condition 11) {
statement 11; //executes in case condition 11 is true
}
else {
statement 12; //executes in case condition 11 is false
}  }
else if(condition 2) {
statement 2; //executes  in case condition 2 is true
}
.
.
.
else {
statement 3; //executes  in case all the conditions are false
}
```

Decision-Making Statements

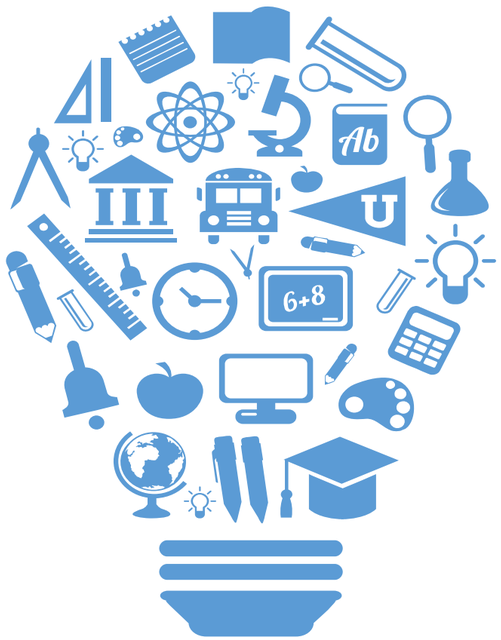
Example of nested **if-else-if** statements:

```
public class Student{
public static void main (String[] args) {
int rank = 20;
if(rank<10) {
if(rank=1) {
System.out.println("You are the topper of this class");
}
else {
System.out.println("You are in 1st section");
}

}
.
.
.
```

Decision-Making Statements

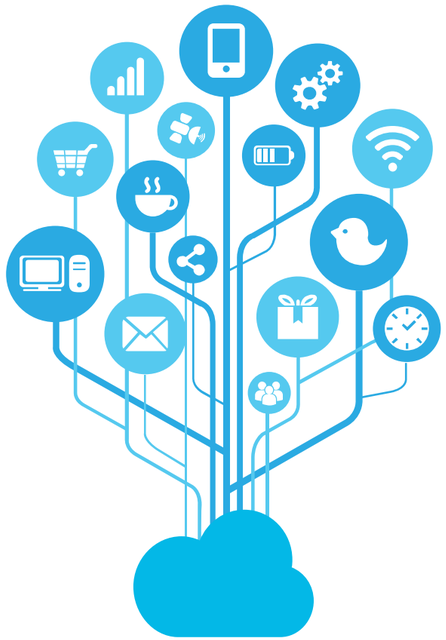
```
.
.
.
else if(rank>=10 && rank<20) {
System.out.println("You are in 2nd section");
}
else if(rank>=20 && rank<30 ) {
System.out.println("You are in 3rd section");
}
else {
System.out.println("You have failed");
}
}
}
```

Decision-Making Statements



- *Switch* statement contains multiple cases which are blocks of codes.
- Single case is executed depending on the variable which is switched.
- int, short, byte, char, and enumeration are supported data types for case variable.
- Cases cannot be duplicate.

Decision-Making Statements



- An optional default statement is executed when none of the cases match the expression.
- Optional break statement is used to terminate the switch block when one case's condition is satisfied.

Decision-Making Statements

Syntax for using nested **switch** statements:

```
switch (expression) {

case value1 {
statement1; //executes in case value1 equals expression
}break;
.
.
.

case valueN {
statementN ; //executes in case valueN equals expression
}break;

}
```

Decision-Making Statements

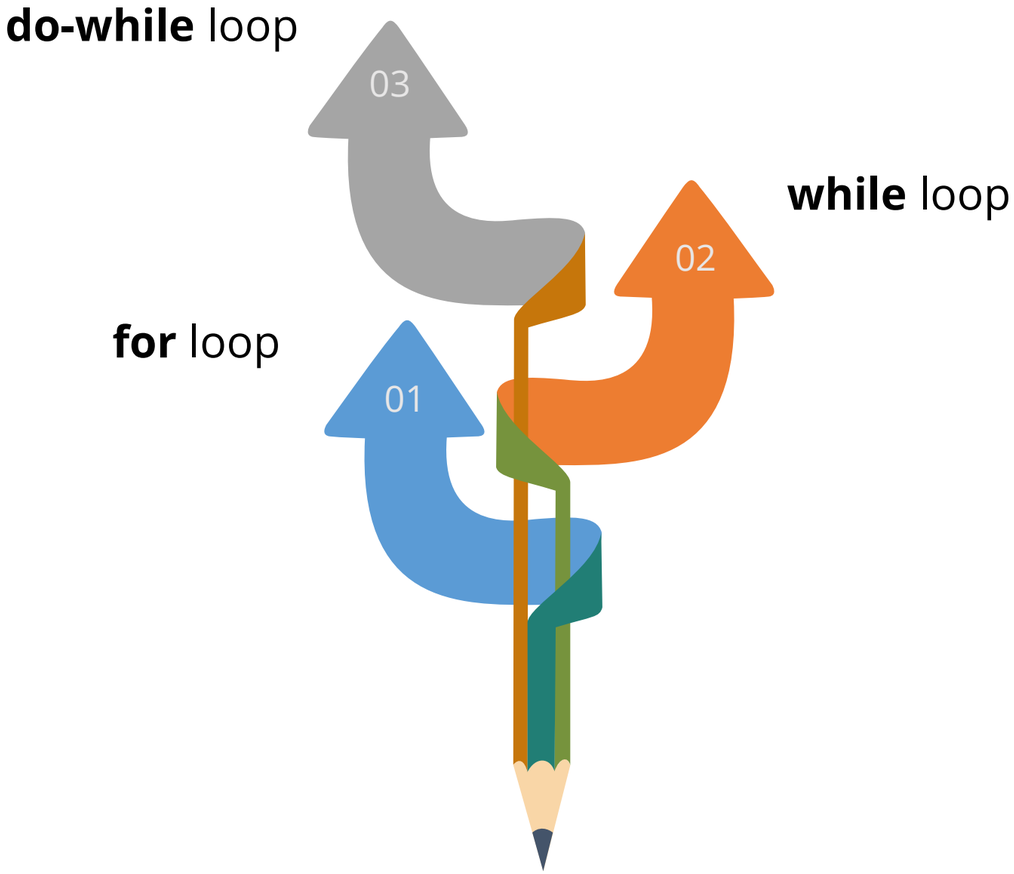
Example of **switch** statements:

```
public class Student{
public static void main (String[] args) {
int rank = 1;
switch (rank)
{
case 1:
System.out.println("You are topper"); break;

case 2:
System.out.println("You rank is 2nd"); break;

default:
System.out.println("Better luck next time");
}
}
}
```

Loop Statements

Loop Statements

Loop statements are used to execute a block of code multiple times. Their execution depends on a particular condition. There are three types of loop statements in Java:



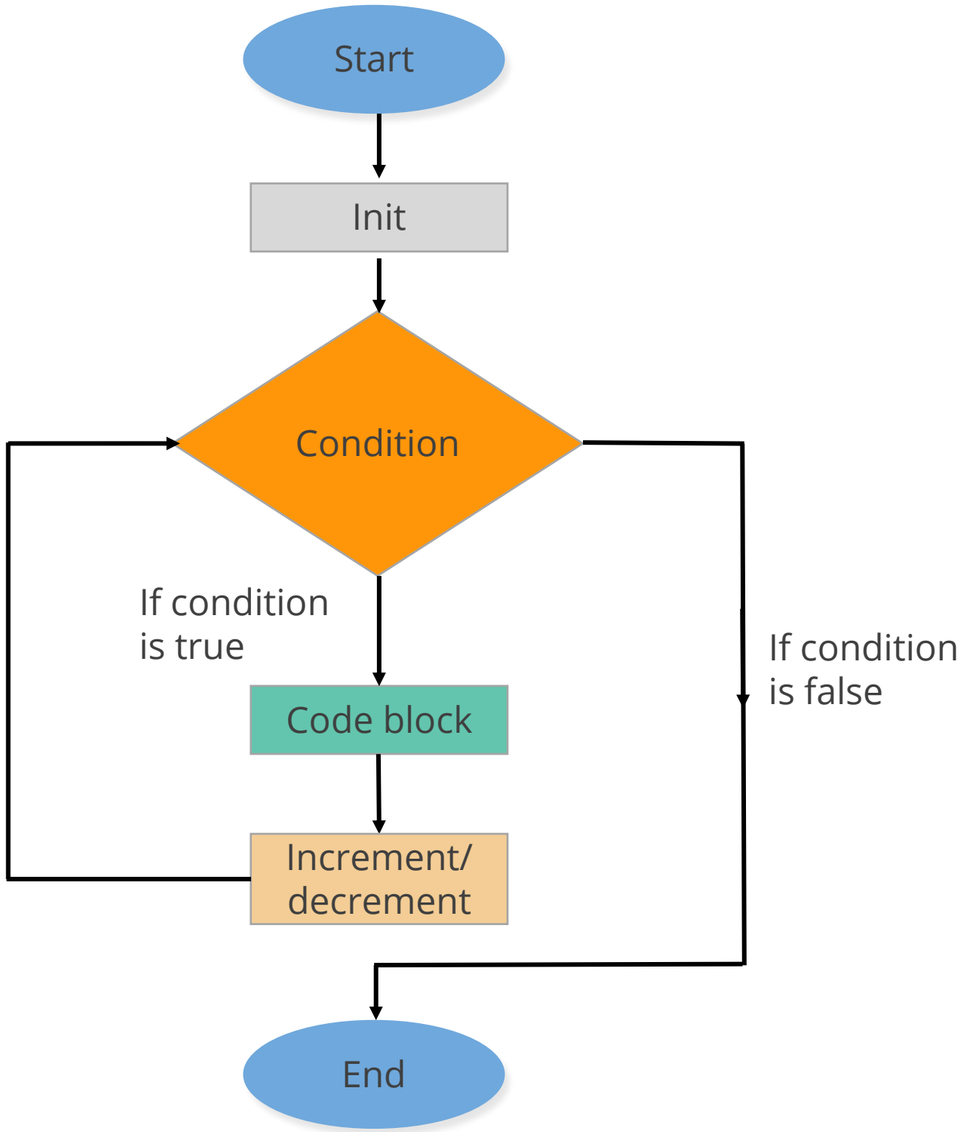
Loop Statements : for Loop

- It is used to iterate a part of the program several times till a condition is met.
- If the number of iterations is fixed, then the **for** loop is recommended to use.

Syntax for using **for** loop:

```
for (initialization; condition; increment/decrement) {

//code statements to be executed

}
```

Loop Statements : for Loop



Loop Statements : for Loop

Example of **for** loop:

```
public class ForExample{

public static void main(String args[]) {

for(int x=0; x<=5; x++){
            System.out.println(x);
        }
    }
}
```

Output:
0
1
2
3
4
5

```
public class ForExample{

public static void main(String args[]) {

for(int x=5; x>=0; x--){
            System.out.println(x);
        }
    }
}
```

Output:
5
4
3
2
1
0

Loop Statements : for Loop

- **for-each** or **enhanced for** loop is a type of **for** loop, which is used on an array or collection.
- Increment or decrement statement is not required here as by default the loop will traverse through all the elements.

Syntax for using **for-each** loop:

```
for (data_type : array_name;) {

//code statements to be executed

}
```

Loop Statements : for Loop

Example of **for-each** loop:

```
public class ForEach{
    public static void main(String[] args){
        //Array declaration
        int ar[]={1,2,3,5,7,11};
        //Using for-each loop to print the array
        for(int x:ar){
            System.out.println(x);
        }
    }
}
```

Output:

```
1
2
3
5
7
11
```

Loop Statements : while Loop

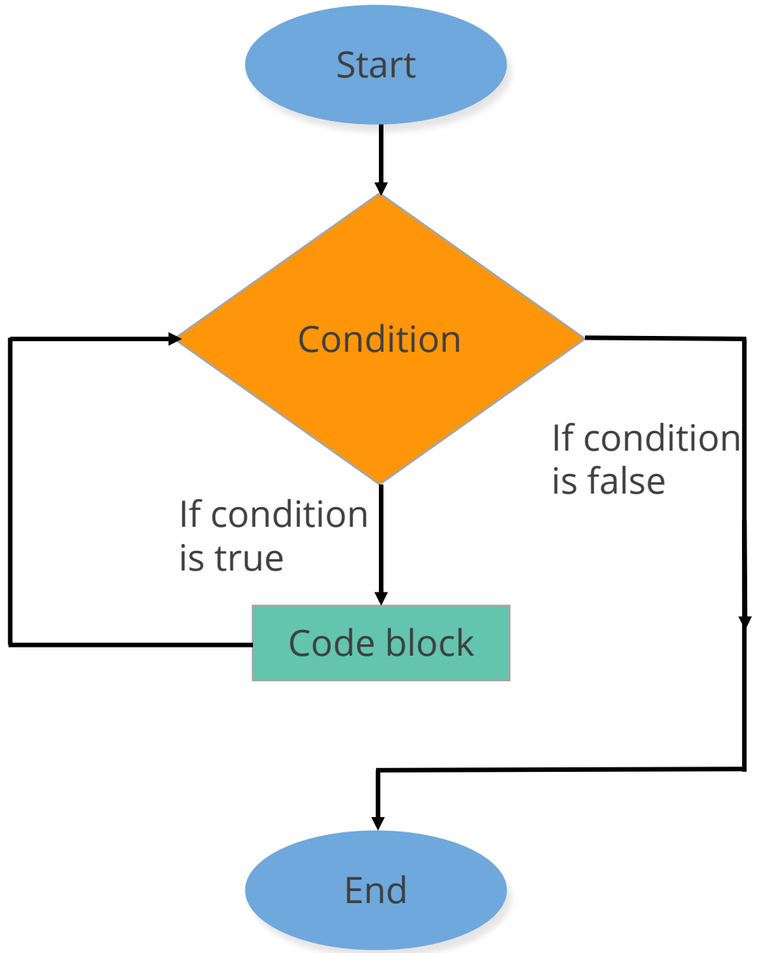
- Like **for** loop, it is used to iterate a part of the program several times till a condition is met.
- If the number of iterations is not fixed, then the **while** loop is preferred.

Syntax for using **while** loop:

```
while (condition) {

//code statements to be executed

}
```

Loop Statements : while Loop

Flow chart of **while** loop:



Loop Statements : while Loop

Examples of **while** loop:

```
public class WhileLoppExample {
 public static void main(String args[]) {
  int i=1;
    while (i<=6) {
    System.out.println("value of i : " + i );
    i++;
    }
   }
}
```

Output:

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

```
public class WhileLoppExample {
 public static void main(String args[]) {
  int i=6;
    while (i>=1) {
    System.out.println("value of i : " + i );
    i--;
    }
   }
}
```

Output:

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

Loop Statements : do-while Loop

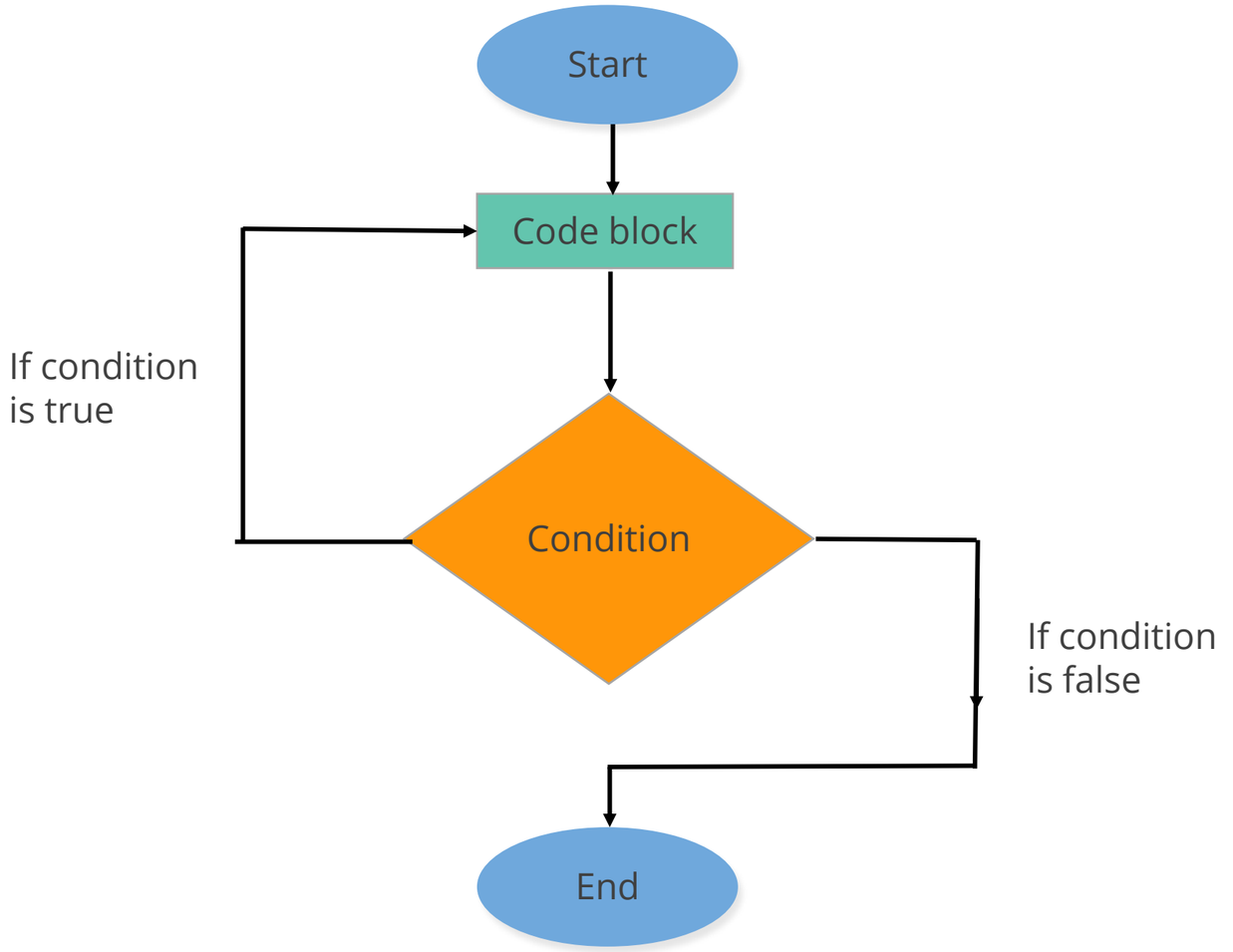
- The **do-while** loop is used to iterate a part of the program several times till a condition is met.
- If the number of iterations is not fixed and the loop needs to be executed at least once, then the **do-while** loop is preferred.
- It is also called the exit control loop as it checks the condition at the end of the loop.

Syntax for using **do-while** loop:

```
do{

//code statements to be executed

} while (condition);
```

Loop Statements : do-while Loop

Flow chart of **do-while** loop:



Loop Statements : do-while Loop

Examples of **do-while** loop:

```
public class DoWhileExample {
public static void main(String[] args) {
    int i=1;
    do{
        System.out.println(i);
    i++;
    }while(i<=6);
}
}
```

Output:

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

```
public class DoWhileExample {
public static void main(String[] args) {
    int i=6;
    do{
        System.out.println(i);
    i--;
    }while(i<=1);
}
}
```

Output

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

Loop Statements : Comparison

Parameter	for Loop	while Loop	do-while Loop
Function	It is a control flow statement that iterates a part of the program multiple times.	It is a control flow statement that executes a part of the programs repeatedly based on a given boolean condition.	It executes a part of the program at least once, and the further iteration is based on the given boolean condition.
When to use?	If the number of iterations is fixed	If the number of iterations is not fixed	If the number of iterations is not fixed, and the loop needs to execute at least once

Loop Statements : Comparison

Parameters	for Loop	while Loop	do-while Loop

Syntax	for(init;condition;incr/decr){
	// code block to be executed
	}
		while(condition){
		//code block to be executed
		}
			do{
			//code block to be executed
			}while(condition);

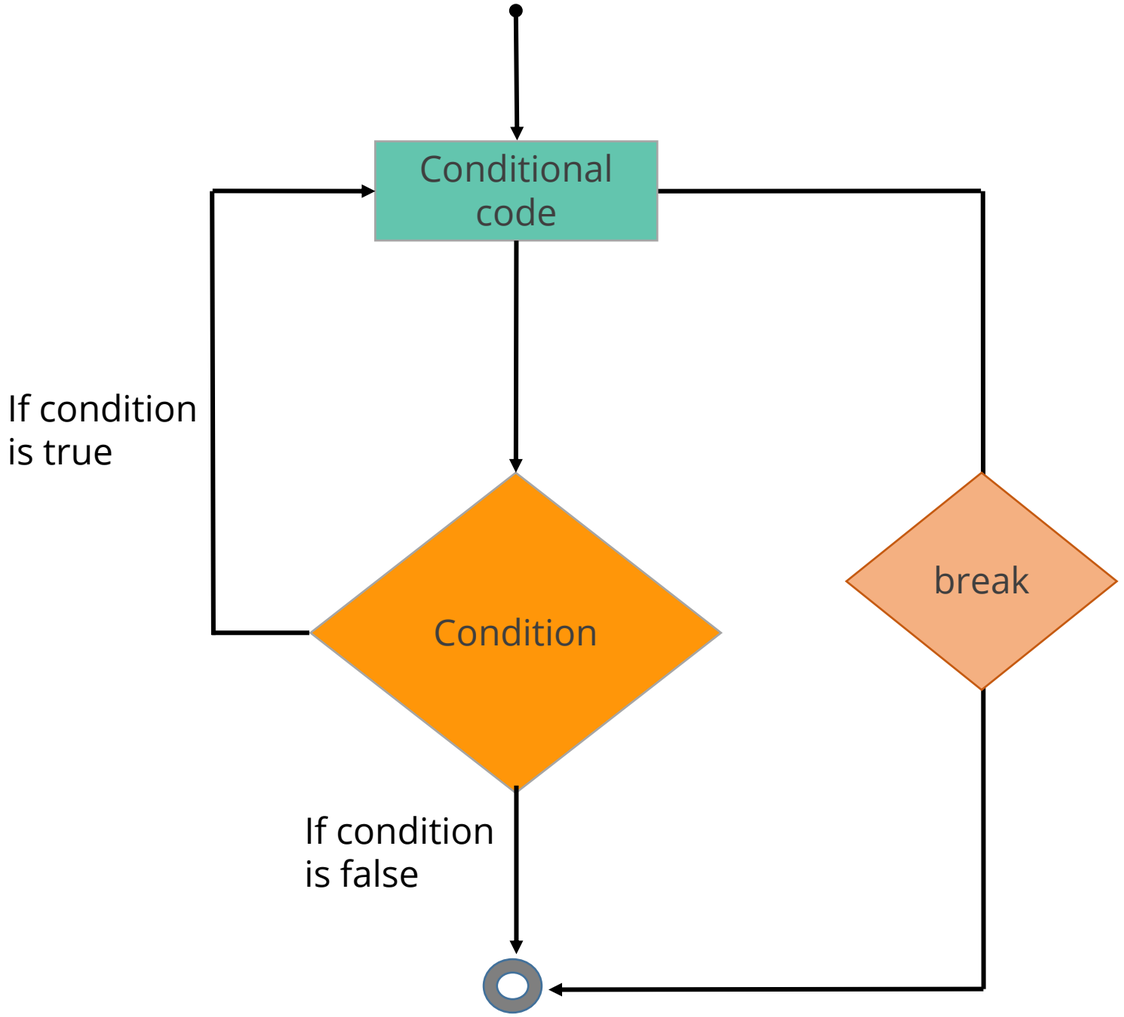
Syntax for infinitive loop	for(;;){
	//code block to be executed
	}
		while(true){
		//code block to be executed
		}
			do{
			//code block to be executed
			}while(true);

Jump Statements

Jump statements or transfer statements are used to transfer control to another part of the program.

Three types of jump statements

Jump Statements : break



- Allows premature exit from a particular block of code
- Commonly used to terminate a sequence in *switch* statements, and to exit a loop
- Syntax: **break**;

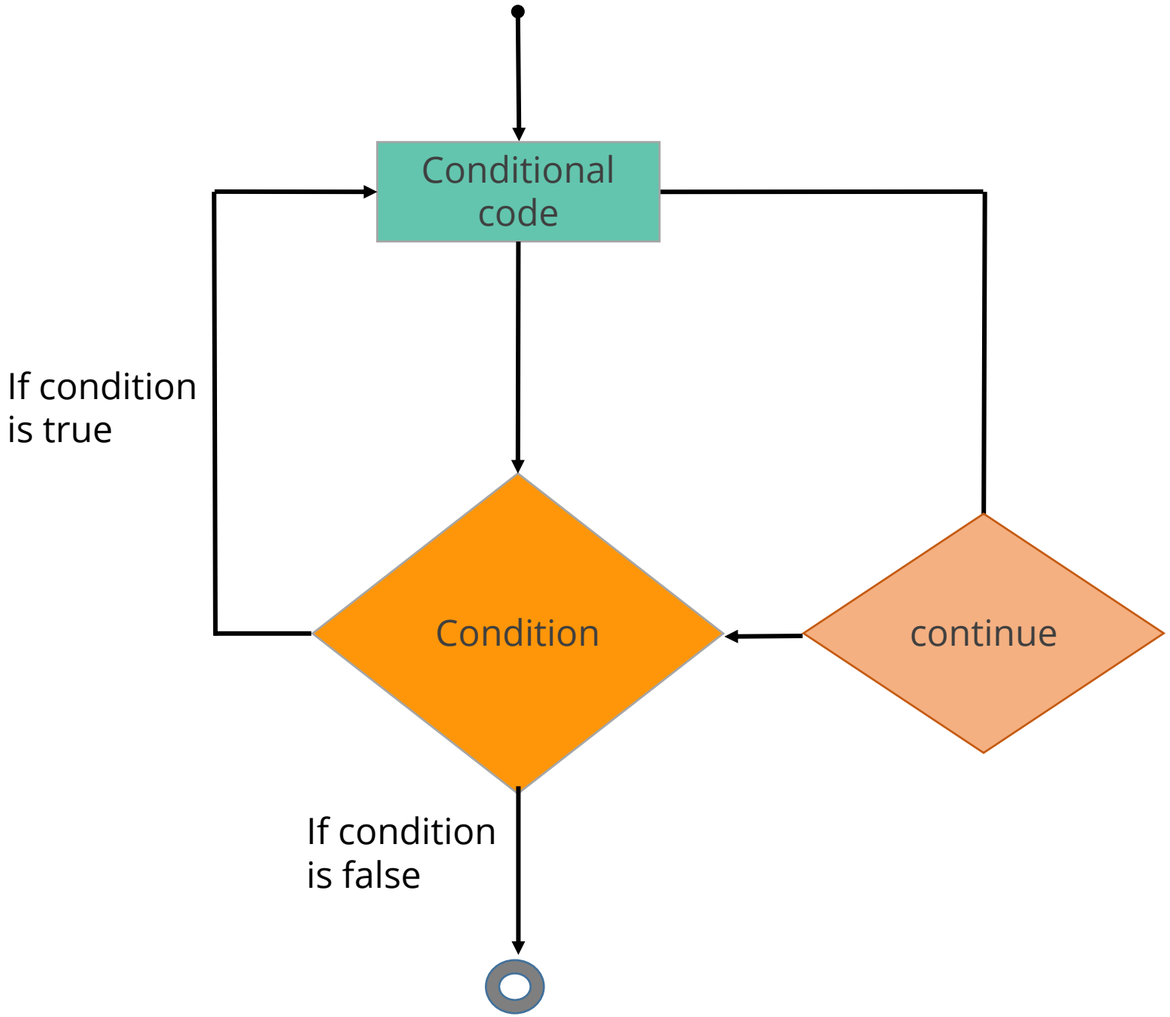
Jump Statements : break

Example of **break** statement:

```
public class Test {

   public static void main(String args[]) {
      int [] numbers = {5, 10, 15, 20, 25};

      for(int x : numbers ) {
         if( x == 20 ) {
            break;
         }
         System.out.print( x );
         System.out.print("\n");
      }
   }
}
```

Output:
5
10
15

Jump Statements : continue



- Allows to skip the remainder of the loop and jump to its end
- Commonly used to force an early iteration of a loop
- Syntax: **continue**;

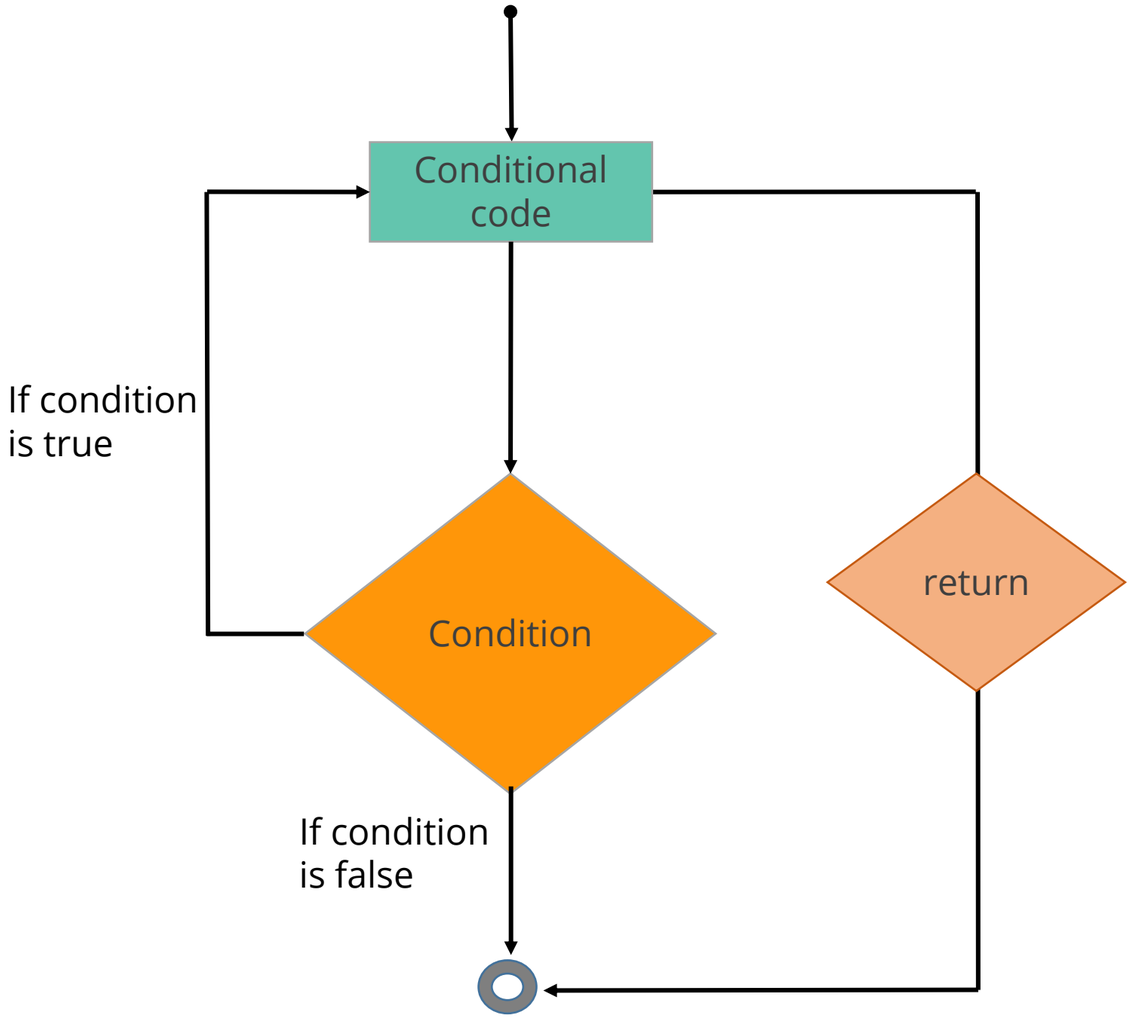
Jump Statements : continue

Example of **continue** statement:

```
public class Test {

   public static void main(String args[]) {
      int [] numbers = {5, 15, 25, 35, 45};

      for(int x : numbers ) {
         if( x == 25 ) {
            continue;
         }
         System.out.print( x );
         System.out.print("\n");
      }
   }
}
```

Output:
5
15
35
45

Jump Statements : return



- Transfers control of the program to the caller method
- Breaks the further execution of the loop when used inside it
- Used for returning a value when the execution of the block is completed
- Syntax: **return**;

Jump Statements : return

Example of **return** statement:

```
class Return {
    public static void main(String args[])
    {
        boolean t = true;
        System.out.println("return Executed.");

        if (t)
            return;

        // Compiler will bypass every statement
        // after return
        System.out.println("Not Executed.");
    }
}
```

Output:
return Executed.

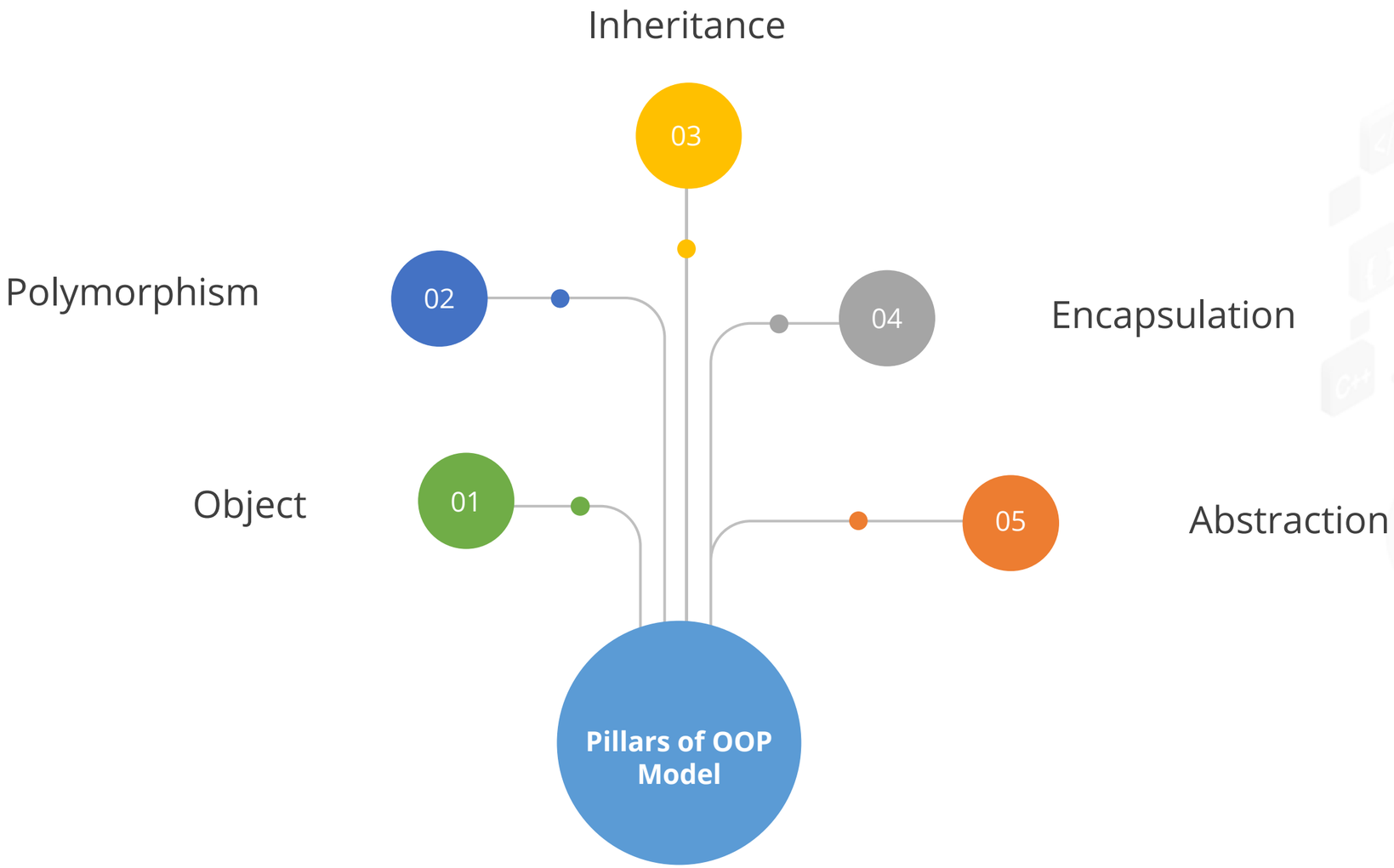
Object-Oriented Programming (OOPs)

Object-Oriented Programming (OOPs)

- OOP is a programming language model, organized around “objects” rather than “actions” and “data” rather than “logic.”
- It simplifies software development and maintenance by providing the following important concepts:
  - Object
  - Polymorphism
  - Inheritance
  - Encapsulation
  - Abstraction

Object-Oriented Programming (OOPs)

Pillars of Object-Oriented Programming model:

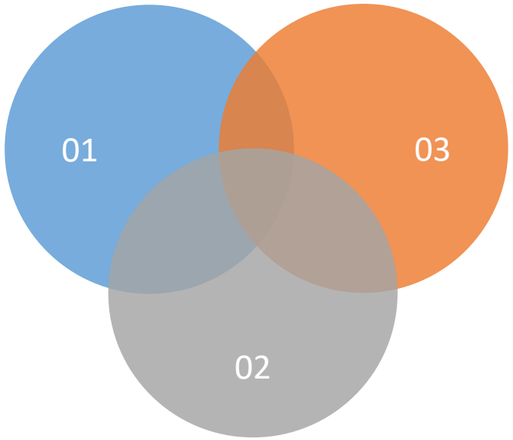


Object-Oriented Programming (OOPs) : Object

- An object is an instance of a class (blueprint).
- It has three characteristics; state, behavior, and identity.
- It can be created using the method named **new()**.

Object-Oriented Programming (OOPs) : Polymorphism

It is the ability of an object to take many forms.



It can be performed using overloading and overriding.

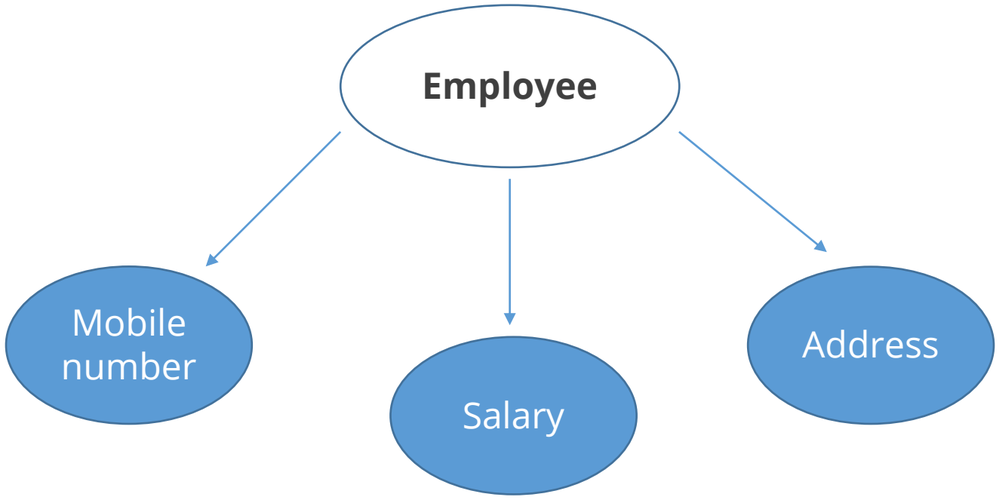
Polymorphism can be of two types: compile time and runtime.

Object-Oriented Programming (OOPs) : Inheritance

Inheritance provides a natural mechanism for organizing and structuring classes, using which subclasses:

- Inherit attributes and services from their superclass
- May add new attributes and services
- Reuse the code in the superclass
- Provide specialized behaviors (overriding and dynamic binding)
- Partially define and implement common behaviors (abstract)

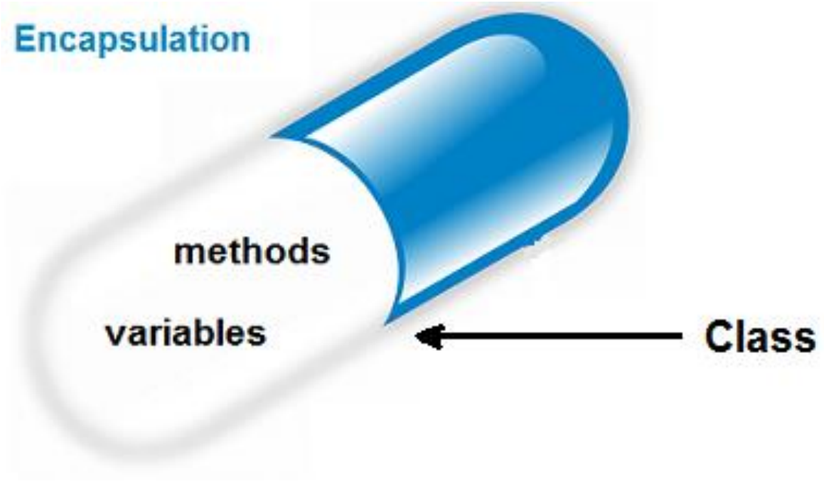
Object-Oriented Programming (OOPs) : Abstraction



Abstraction

- Hiding internal details and showing functionality is known as abstraction.
- For example an employee of a company with hidden mobile number, salary, and address.
- In Java, we use abstract class and interface to achieve abstraction.

Object-Oriented Programming (OOPs) : Encapsulation



- Encapsulation is a mechanism for wrapping the variables and methods together as a single unit.
- In encapsulation, variables from a class will be hidden from other outer classes and can only be accessed by the method of the same class.
- Using encapsulation, a field in a class can be made read-only or read and write only via the methods and not directly

Packages

Packages

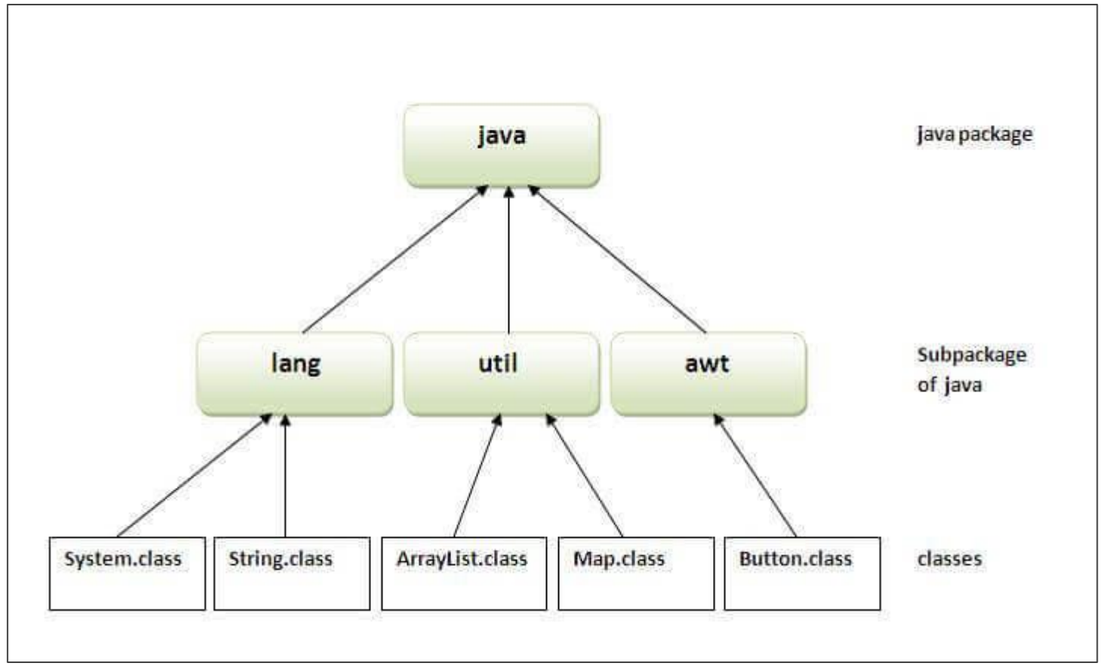
- A Java *package* is a collection of sub-packages, interfaces, and classes of a common type.
- They can be of two types:

  Built-in

  User-defined
- The keyword *package* is used to create **packages** in Java.
- Some of the important built-in **packages** are java, lang, awt, javax, swing, net, io, util, sql etc.
- For command prompt, following is the syntax to compile a Java package:

  javac -d directory javafilename

Packages



Advantages of a package:

- A package helps in maintaining a categorization of classes and interfaces.
- It helps in providing access protection by means of the required access modifier.
- It also removes any possibility of naming collision during compilation of the Java file.

Image Source - https://www.javatpoint.com/package

Packages

Example of a **package**:

```
package mypackage;
public class Pack{
 public static void main(String args[]){
    System.out.println("Welcome to mypackage");
   }
}
```

Output:

```
Welcome to mypackage
```

Packages

There are three ways to access a **package**:

1. Using *import package.*;*

```
//save by First.java
package pack1;
public class First{
  public void msg1(){System.out.println("Hello First");}
}

//save by Second.java
package pack2;
import pack1.*;

class Second{
  public static void main(String args[]){
   First obj1 = new First();
   obj1.msg();
  }
}
```

Output: Hello First

Packages

2. Using *import package.classname;*

```
//save by First.java
package pack1;
public class First{
  public void msg1(){System.out.println("Hello First");}
}

//save by Second.java
package pack2;
import pack1.First;

class Second{
  public static void main(String args[]){
   First obj1 = new First();
   obj1.msg();
  }
}
```

Output: Hello First

Packages

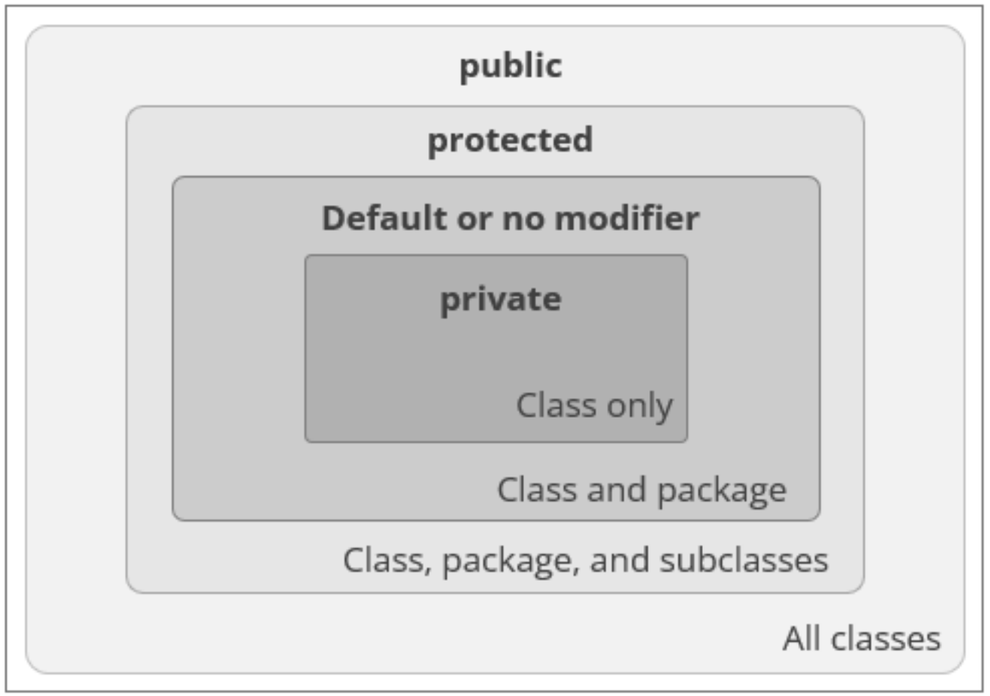
3. Using fully qualified name

```
//save by First.java
package pack1;
public class First{
  public void msg1(){System.out.println("Hello First");}
}

//save by Second.java
package pack2;

class Second{
  public static void main(String args[]){
   pack1.First obj1 = new pack1.First();
   obj1.msg();
  }
}
```

Output: Hello First

Access Modifiers

Access Modifiers



- Access modifiers set access levels for classes, variables, methods, and constructors.
- They have four levels:
  - Private
  - Default
  - Protected
  - Public
- Syntax:
  accessModier class/constructor/variable/method

Access Modifiers

Example of a **public** access modifier:

```
public class Car{

        /*Code Statements*/
  }
```


The **public** access modifier has the widest scope amongst all the parameters. It is accessible everywhere.

Access Modifiers

Example of a **protected** access modifier:

```
public class Car{

        protected void mesg(){
        System.out.println("Accelerate");
        }

  }
```


The **protected** access modifier is accessible within and outside the package but through inheritance only.

Access Modifiers

Example of a **private** access modifier:

```
public class Car{

        private int data=20;

        protected void mesg(){
        System.out.println("Accelerate");
        }

  }
```


The **protected** access modifier is accessible within and outside the package but through inheritance only.

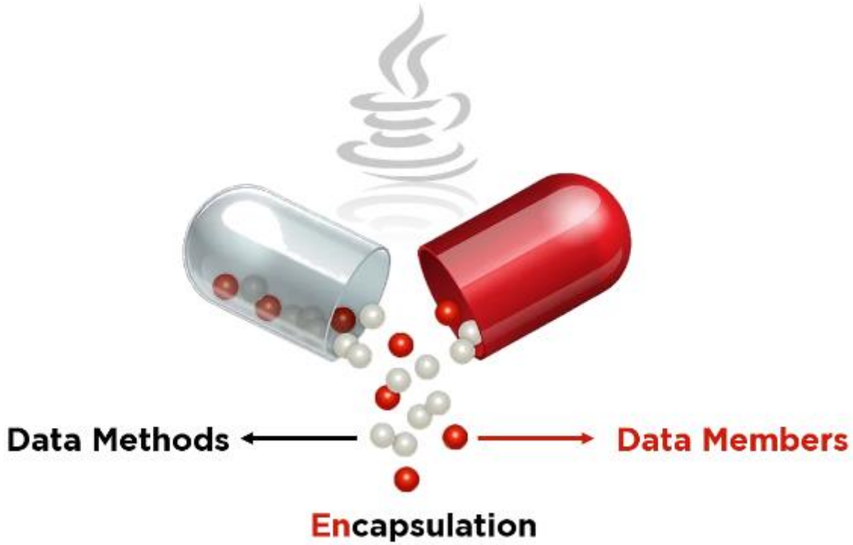
Access Modifiers

Comparison amongst all access modifiers:

Access Modifier	Within Class	Within Package	Outside Package (by child class only)	Outside Package
Private	**Yes**	No	No	No
Default	**Yes**	**Yes**	No	No
Protected	**Yes**	**Yes**	**Yes**	No
Public	**Yes**	**Yes**	**Yes**	**Yes**

Encapsulation

Encapsulation



- Encapsulation in Java refers to integrating data (variables) and code (methods) into a single unit.
- In encapsulation, a class's variables are hidden from other classes and can only be accessed by the methods of the class in which they are found.

Image Source - https://www.simplilearn.com/tutorials/java-tutorial/java-encapsulation#:~:text=Encapsulation%20in%20Java%20refers%20to,in%20which%20they%20are%20found.

Encapsulation

Syntax while using encapsulation:

```
<Access_Modifier> class <Class_Name>
{
 private <Data_Members>;
 private <Data_Methods>;
}
```

Encapsulation

Example of encapsulation:

```
package dc;
public class c
{
public static void main (String[] args)
{
Employee e = new Employee();
e.setName("Robert");
e.setAge(33);
e.setEmpID(1253);
System.out.println("Employee's name: " +
e.getName());
System.out.println("Employee's age: " + e.getAge());
System.out.println("Employee's ID: " + e.getEmpID());

}
}
```

```
package dc;
public class Employee {
private String Name;
private int EmpID;
private int Age;
public int getAge() {
return Age;
}
public String getName() {
return Name;
}
public int getEmpID() {
return EmpID;
}
public void setAge(int newAge) {
Age = newAge;
}
```

Encapsulation

Example of encapsulation:

```
public void setName(String newName) {
Name = newName;
}
public void setRoll(int newEmpID) {
EmpID = newEmpID;
}
public void setEmpID(int EmpID) {
}
}
```

Output:

```
Employee's name: Robert
Employee's age: 33
Employee's ID: 1253
```

Encapsulation

Advantages of encapsulation:

1. Better Control

Encapsulation provides ultimate control over the data members and data methods inside the class.

2. Getter and Setter

The standard IDEs provide in-built support for ‘Getter and Setter’ methods, which increases the programming pace.

Encapsulation

Advantages of encapsulation:

3. Security

It prevents access to data members and data methods by any external classes, which improves the security of the encapsulated data.

4. Flexibility

Changes made to one part of the code can be successfully implemented without affecting any other part of the code.

Inheritance

Inheritance

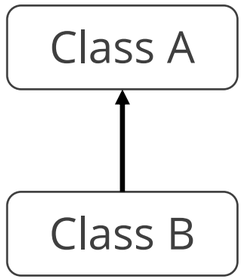
Inheritance provides a natural mechanism for organizing and structuring classes, using which subclasses (child classes) inherit all the properties (states) and behaviors from their superclasses.

Syntax for using inheritance:

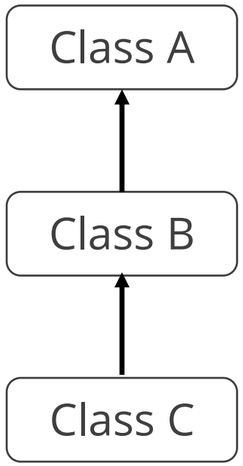
```
class SubClass_name extends ParentClass_name
{
//DataMembers;
//Methods;
}
```

Inheritance

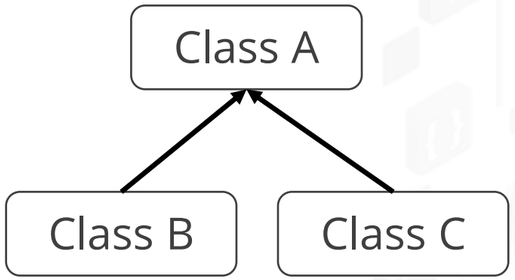
Types of inheritance:



1. Single-level



2. Multilevel



3. Hierarchical

Inheritance

1. Single-level inheritance:

```
class Student {
void Play() {
System.out.println("Playing Cricket...");
}
}
class Bob extends Student {
void Study() {
System.out.println("Studying Science...");
}
}
public class Single {
public static void main(String args[]) {
Bob d = new Bob();
d.Study();
d.Play();
}
}
```

Output:

```
Studying Science...
Playing Cricket...
```

Inheritance

2. Multilevel inheritance:

```
class Mammal{
void eat(){System.out.println("eating...");}
}
class Lion extends Mammal{
void growl(){System.out.println("growling...");}
}
class Cub extends Lion{
void weep(){System.out.println("weeping...");}
}
class Test{
public static void main(String args[]){
Cub d=new Cub();
d.weep();
d.growl();
d.eat();
}}
```

Output:

```
weeping...
growling...
eating...
```

Inheritance

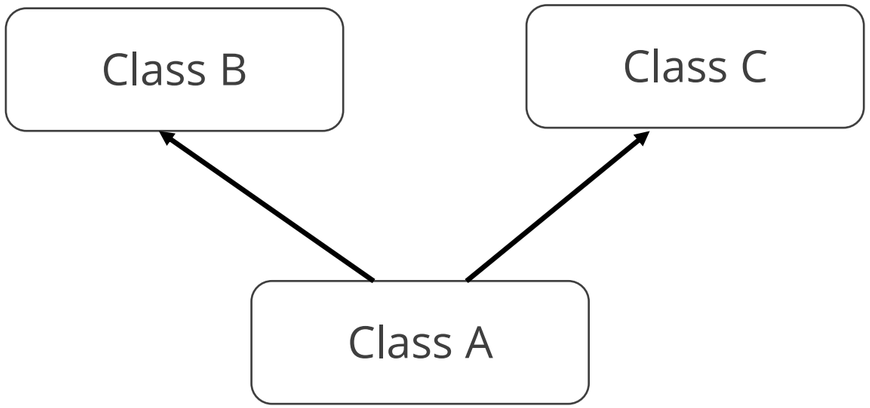
3. Hierarchical inheritance:

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

Lion l=new Lion();
l.growl();
l.eat();

}}
```

Output:

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

Inheritance



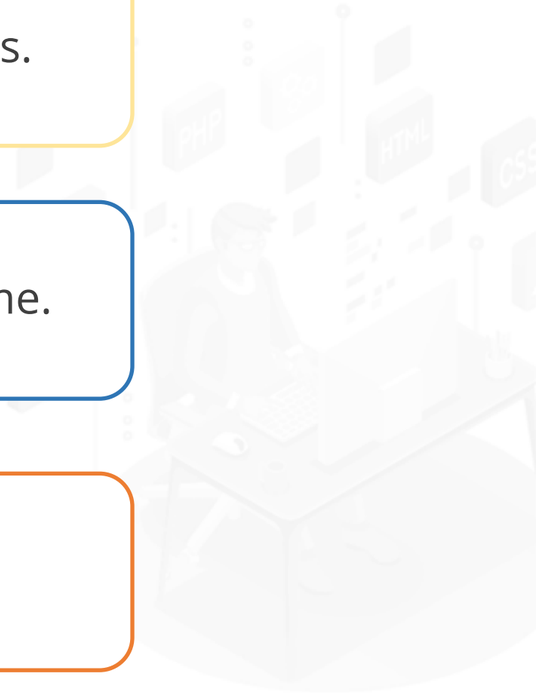
Multiple Inheritance

- Multiple inheritance is *not supported* in Java as it may cause ambiguities.
- In case both the parent classes have a method with the same name and the child class object calls the same, there will be an ambiguity.

Polymorphism

Polymorphism

- Polymorphism is the ability of an object to take many forms.
- Polymorphism can be of two types: compile time and runtime.
- It can be performed using overloading and overriding.



Polymorphism : Method Overloading

- Method overloading is a feature that allows a class to have two or more methods that have the same name but different arguments.
- Argument lists can differ in factors such as the number of parameters and data types.

Two ways to perform method overloading:

- By changing the number of arguments
- By changing the data type of variables

Polymorphism : Method Overloading

Example of method overloading by changing the number of arguments:

```
public class Operation{
static int add(int a,int b){
        return a+b;
        }
static int add(int a,int b,int c){
        return a+b+c;
        }

public static void main(String[] args){
System.out.println(add(11,11));
System.out.println(add(11,11,11));

}
}
```

Output:

```
22
33
```

Polymorphism : Method Overloading

Example of method overloading by changing the data types:

```
public class Operation{
static int add(int a,int b){
        return a+b;
        }
static int add(double a,double b){
        return a+b;
        }

public static void main(String[] args){
System.out.println(add(11,11));
System.out.println(add(11.2,11.2));

}
}
```

Output:

```
22
22.4
```

Polymorphism : Method Overriding

- In Java, method overriding happens if a child class has the same method as the parent class method.
- It is used to provide a specific implementation of a method of a super (parent) class.

Rules of method overriding:

- Child Class method must have the same name as the parent class method.
- Child Class method must have the same parameters as the parent class method.

Polymorphism : Method Overriding

Example of method overriding:

```
class Vehicle{
  //defining a method
  void run(){System.out.println("Vehicle is moving");}

}

class Car extends Vehicle{
    void run(){
        System.out.println("Car is moving");
        }

  public static void main(String args[])
        {
  Car obj = new Car();
  obj.run();
  }
}
```

Output:
Car is moving

Polymorphism : Dynamic Method Dispatch

- DMD or runtime polymorphism is a process in which an overridden method is called through the reference variable of the parent class.
- In runtime polymorphism, the call to the overridden method is resolved at run time instead of compile time.
- It is executed using upcasting process.

Syntax for upcasting:

```
class ParentClassName {}

class ChildClassName extends ParentClassName {}

ParentClassName Reference_variable = new ChildClassName();
```

Polymorphism : Dynamic Method Dispatch

Example of Dynamic Method Dispatch:

```
class Car{
  void run(){System.out.println("running");}
}

class Thar extends Car{
  void run(){System.out.println("running offroad");}

  public static void main(String args[]){
    Car c = new Thar();//upcasting
    c.run();
   }
}
```

Output:
running offroad

Constructors

Constructors

- A constructor is a set of instructions designed to initialize an instance.
- Parameters can be passed to the constructors in the same way as for a method.
- Public, protected, and private are the valid access modifiers for constructors.
- Java compiler creates a default constructor if it doesn't find it at the time of object creation.

Syntax for using **constructors**:

```
accessModifier className ( arguments )
{
/* Code Statements */
}
```

Constructors : Rules

Rules for creating constructors:

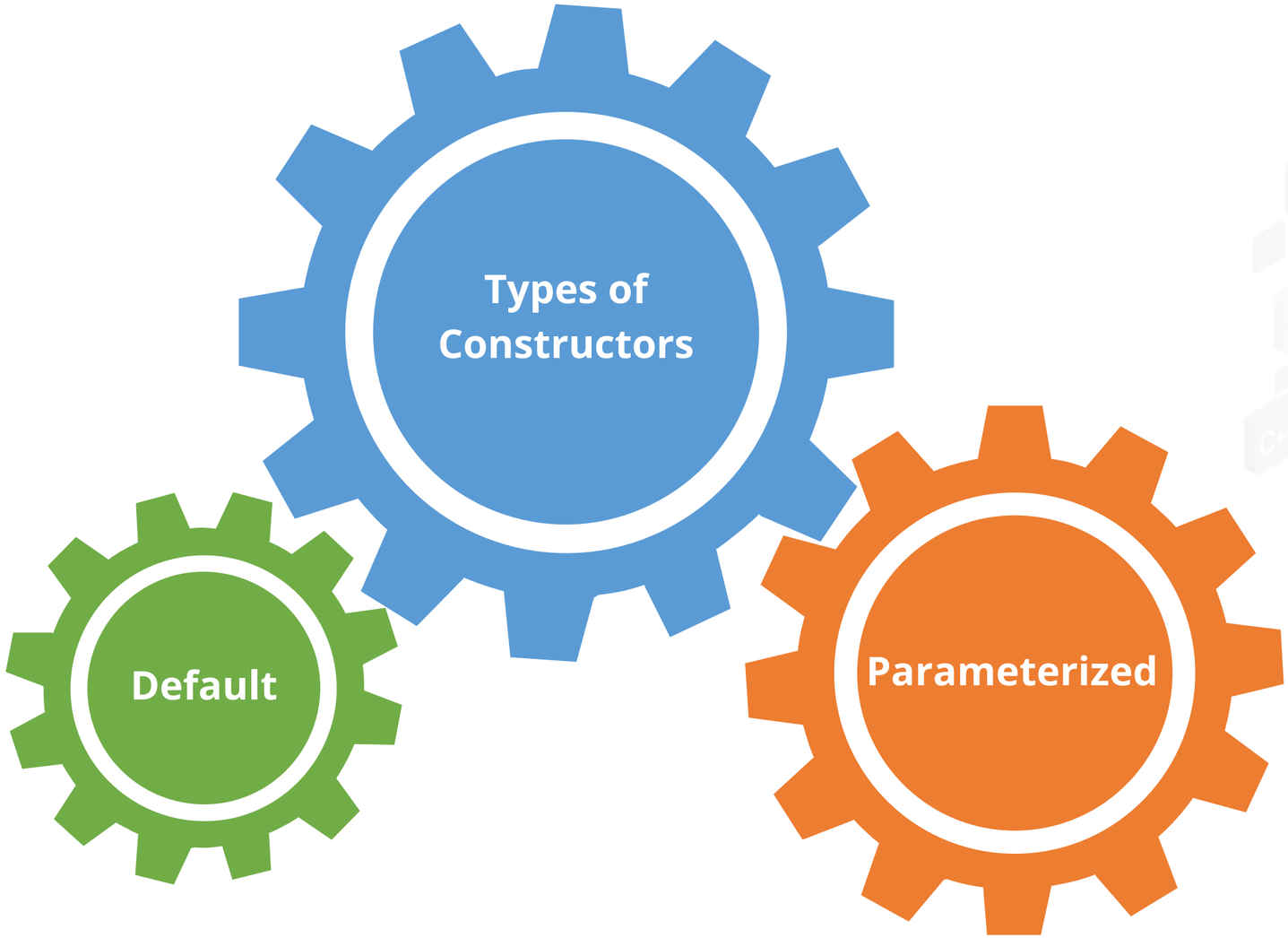
01 A constructor’s name must be the same as that of the class.

02 It should have no explicit return type.

03 A constructor cannot be static, abstract, final, and synchronized.

Constructors : Types

There are two types of constructors:



Constructors : Default

- A constructor with no argument is called a default constructor.
- It enables you to create object instances with “**new class_name()**”.
- A user-added constructor declaration to a class overwrites the default constructor.
- It provides default values like zero, null, etc., to the object.
- It’s syntax is “**class_name () {}**”.

Constructors : Default

Example of a **default** constructor:

```
class Yamaha{
//creating a default constructor
Yamaha(){System.out.println("Yamaha is created");}
//main method
public static void main(String args[]){
//calling a default constructor
Yamaha b=new Yamaha();
}
}
```

Output: Yamaha is created

Constructors : Parameterized

- In case a constructor has any specific parameter, it is called a parameterized constructor.
- It enables you to create object instances with "**new class_name(arg_list)**".
- It is used to provide different values to each of the objects created.
- Its syntax is:

```
class_name (arg_list) {
        /*Respective Code Statements*/
}
```

Constructors : Parameterized

Example of a **parameterized** constructor:

```
class RoyalEnfield{
    int num;
    String color;
    //creating a parameterized constructor
    RoyalEnfield(int i,String c){
    num = i;
    color = c;
    }
    //method to display the values
    void display(){System.out.println(id+" "+color);}

    public static void main(String args[]){
    //creating objects and passing values
     RoyalEnfield  r1 = new  RoyalEnfield(111,"Black");
     RoyalEnfield  r2 = new  RoyalEnfield(222,"Dusty");
    //calling method to display the values of object
    s1.display();
    s2.display();
   }
```

Output:

```
111 Black
222 Dusty
```

Constructors : Overloading

Example of constructor overloading:

```
class RoyalEnfield{
    int num;
    String color;
    int speed;

        //creating a two argumented constructor
    RoyalEnfield(int i,String c){
    num = i;
    color = c;
    }

       //creating a three argumented constructor
    RoyalEnfield(int i,String c, int s){
    id = i;
    color = c;
    speed = s;
    }
```

```
    .
    .

    //method to display the values
     void display(){System.out.println(id+" "+color+"
"+speed);}

    public static void main(String args[]){
    //creating objects and passing values
     RoyalEnfield  r1 = new  RoyalEnfield(111,"Black")
;
       RoyalEnfield  r2  =new  RoyalEnfield(222,"Dusty",
120);
    //calling method to display the values of object
    s1.display();
    s2.display();
   }
```

Output:

```
111 Black 0
222 Dusty 120
```

Constructors : Comparison

Constructors	Methods
It is used to initialize an object's state.	It represents the behavior of an object.
A constructor can not have a return type.	A return type is mandatory.
It is invoked implicitly.	It is invoked explicitly.
The compiler provides with a default constructor if the constructor is not created explicitly.	A method always needs to be created by the programmer.
It must have the same name as that of the class.	Methods need not have the same name as that of the class.

Keywords: static and final

Keyword : static

- **static** is used for memory management in Java.
- It belongs to the class instead of the instance.
- **static** keyword can be applied to :
  - Methods
  - Variables
  - Block
  - Nested Class
- The main() method is a **static** method because the JVM does not create an instance of a class when executing the main method.
- The **static** methods should be accessed using the class name instead of the object name.

Keyword : static

Example of a **static** method:

```
class Student{
     int rollno;
     String name;
     static String college = "JECRC";
     //static method to change the value of static
       variable
     static void change(){
     college = "CTAE";
     }
     //constructor to initialize the variable
     Student(int r, String n){
     rollno = r;
     name = n;
     }
//method to display values
     void display(){System.out.println(rollno+" "+name+"
 "+college);}
}
```

```
     //Test class to create and display the values of
object
public class TestStaticMethod{
    public static void main(String args[]){
    Student.change();//calling change method
    //creating objects
    Student s1 = new Student(111,"Deep");
    Student s2 = new Student(222,"Amit");
    Student s3 = new Student(333,"Shubham");
    //calling display method
    s1.display();
    s2.display();
    s3.display();
    }
}
```

Output:

```
111 Deep CTAE
222 Amit CTAE
333 Shubham CTAE
```

Keyword : final

- **final** is a reserved keyword which stops the value change, method overriding, and inheritance in Java.
- It can be applied to:
  - Methods
  - Variables
  - Classes
- **final** class once declared, cannot be subclassed or inherited.
- **final** method cannot be overridden after declaration.
- **final** variable cannot be changed after being declared.

Keyword : final

Example of a **final** variable:

```
class Yamaha{
  final int speedlimit;//blank final variable

  Yamaha(){
  speedlimit=70;
  System.out.println(speedlimit);
  }

  public static void main(String args[]){
    new Yamaha();
 }
```

Output: Compile time error

Keyword : this

- The “**this”** keyword is a reference variable that is used to refer to the current object.
- It can be used to refer current class instance variable.
- It can also be used to invoke current class method and current class constructor.
- It can be passed as an argument in the calls of a method as well as a constructor.
- It can also be used to return the current class instance from the method.

Keyword : this

Example of **this** keyword:

```
class A{
void original(){System.out.println("hello original");}
void new(){
System.out.println("hello new");
//original();//same as this.originsl()
this.original();
}
}

public class Test{

public static void main(String args[]){
        A a=new A();
        a.new();
}}
```

Output:
hello new
hello original

Keyword : this

Example of **this** keyword:

```
class A{
  A2 obj;
  A(A2 obj){
    this.obj=obj;
  }
  void display(){
    System.out.println(obj.data);
  }
}

class A2{
  int data=10;
  A2(){
   A a=new A(this);
   a.display();
  }
  public static void main(String args[]){
   A2 a2=new A2();
  }
}
```

Output:10

Keyword : super

- The **super** keyword is a reference variable that used to refer immediate parent class object.
- It can be used to refer immediate parent class instance variable.
- It can also be used to invoke immediate parent class method and constructor.

Keyword : super

Example of **super** keyword:

```
class Animal{
String color="white";
}
class Lion extends Animal{
String color="brown";
void printColor(){
System.out.println(color);
System.out.println(super.color);
}
}
class TestingSuper{
public static void main(String args[]){
Lion l=new Lion();
l.printColor();
}}
```

```
Output:
brown
white
```

Keyword : super

Example of **super** keyword:

```
class Animal{
Animal(){System.out.println("Origin: Animal");}
}
class Lion extends Animal{
Lion(){
super();
System.out.println("Origin: Lion");
}
}
class TestingSuper{
public static void main(String args[]){
Lion l=new Lion();
}}
```

Output:

```
Origin: Animal
Origin: Lion
```

Abstract Class and Method

Abstract Class and Method

- Any class in java that is declared abstract is called abstract class.
- It can not be instantiated.
- Syntax:

  abstract class ClassName{}

- A method that does not have an implementation and is declared as abstract is called abstract method.
- Syntax:

  abstract void methodName();

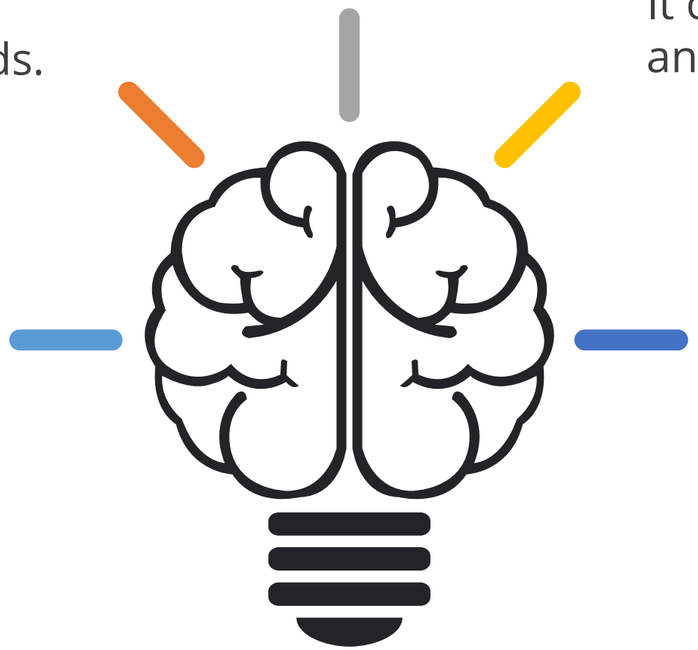
Abstract Class and Method

Rules for creating an abstract class:

Abstract class cannot be instantiated.

It can have both abstract and non-abstract methods.

It can have static methods and constructors.



It must be declared with an abstract keyword.

Abstract class can also have final methods.

Abstract Class and Method

Example of **abstract** class and method:

```
abstract class Car{
   Car(){System.out.println("Car is created");}
   abstract void run();
    void changeGear(){System.out.println("Gear changed");
}
 }
//Creating a Child class which inherits Abstract class
 class Toyota extends Car{
 void run(){System.out.println("Moving safely");}
 }
//Creating a Test class which calls abstract and non-
abstract methods
 class Testing{
 public static void main(String args[]){
  Car obj = new Toyota();
  obj.run();
  obj.changeGear();
 }
}
```

Output:

```
Car is created
Moving safely
Gear changed
```

Interface

Interface

- Interface is a blueprint of a class that has static constants and abstract methods.
- All the methods are abstract by default.
- It is a mechanism to achieve abstraction and multiple inheritance.

Syntax of declaring an interface:

```
interface interfaceName {
  // declare constant fields
  // declare abstract methods
}
```

Interface

Importance of creating an interface:

Used to achieve abstraction

Facilitates multiple inheritance

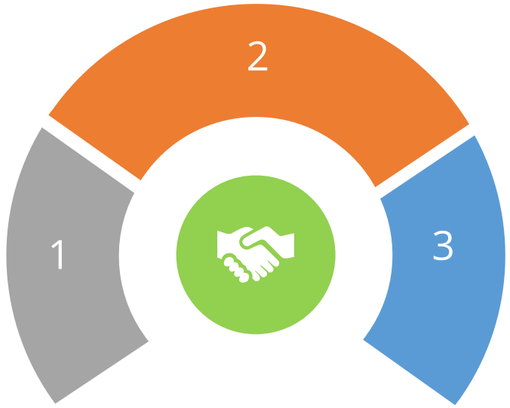
Used to achieve loose coupling

Interface

Relationship between classes and interfaces:

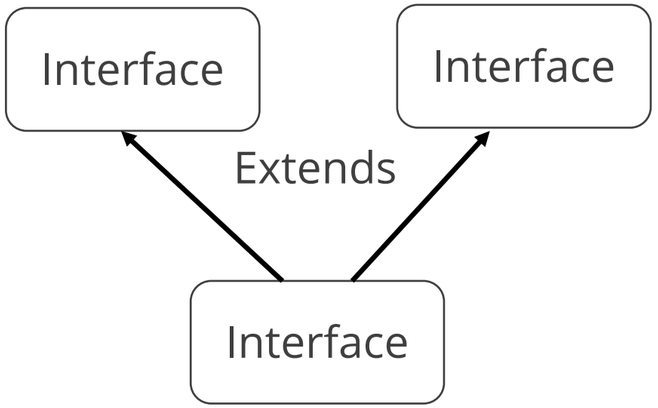
An interface can extend another interface.

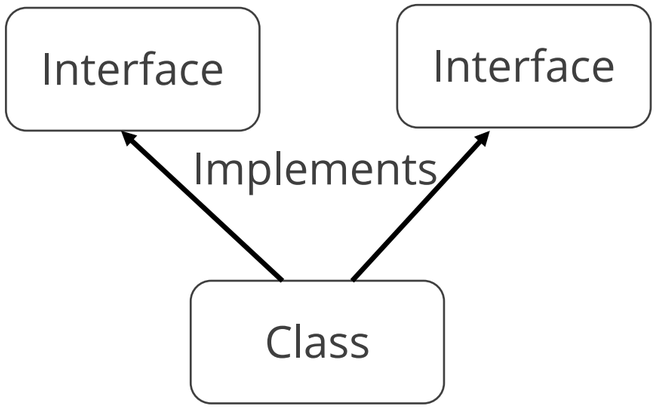
A class can extend another class.



A class **implements** an interface.

Interface





- Multiple inheritance becomes possible if a class implements multiple interfaces, or an interface extends multiple interfaces.
- It does not give a compilation error like a class inheriting multiple classes as the methods are getting implemented by the subclass.

Interface

Example of **interface**:

```
interface Displayable{
void display();
}
interface Exhibition{
void exhibit();
}
class Test implements  Displayable, Exhibition{
public void display(){System.out.println("Displayed");}

public void exhibit(){System.out.println("Exhibited");}


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

Output:

```
Displayed
Exhibited
```

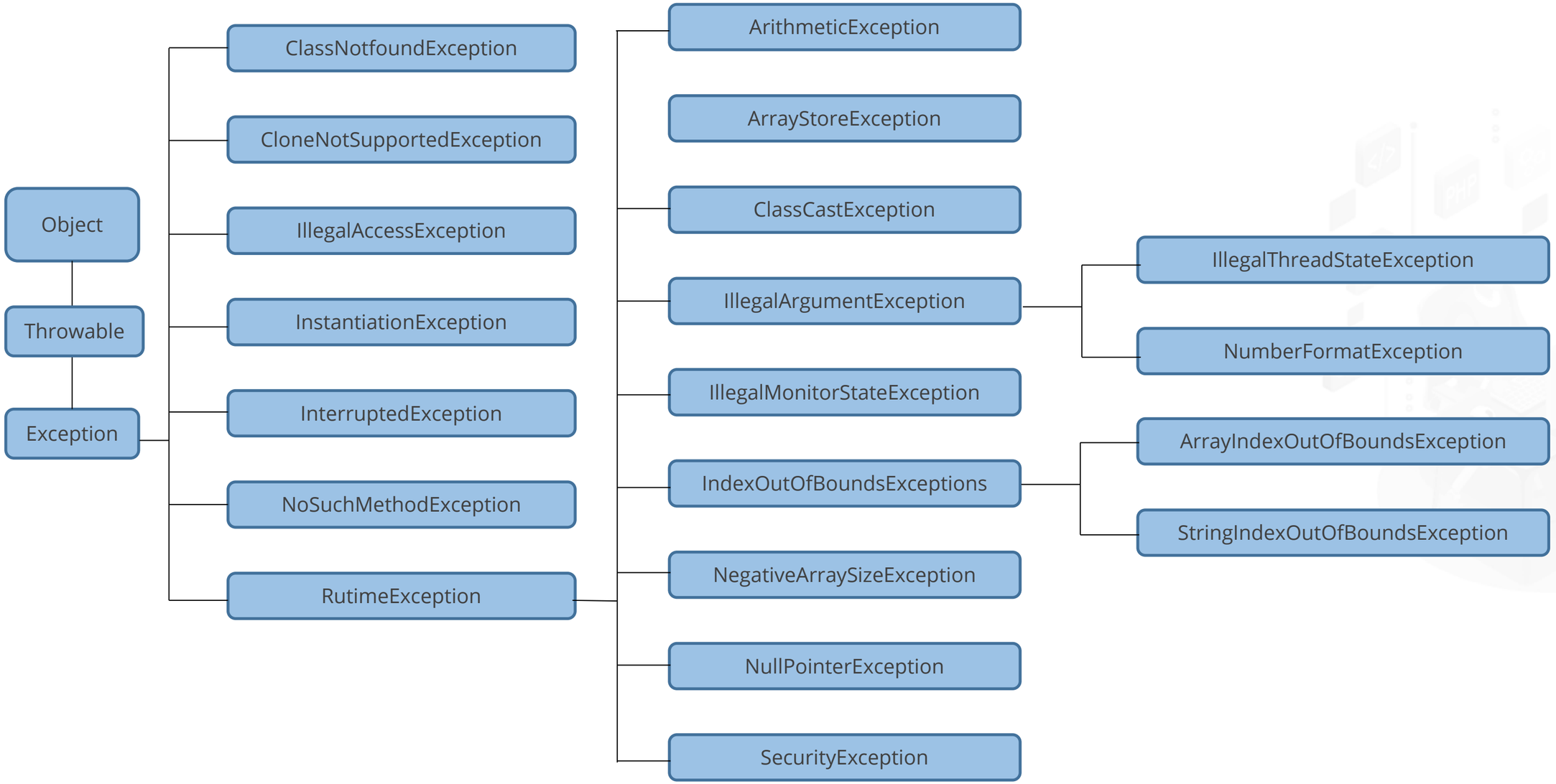
Exception Handling

Exception Handling

- Exception handling is an important concept in Java to handle runtime errors.
- In Java, an exception is an event that can disrupt the flow of the program.
- This event is an object thrown at runtime.
- To maintain the normal flow of the program, exception handling is done.

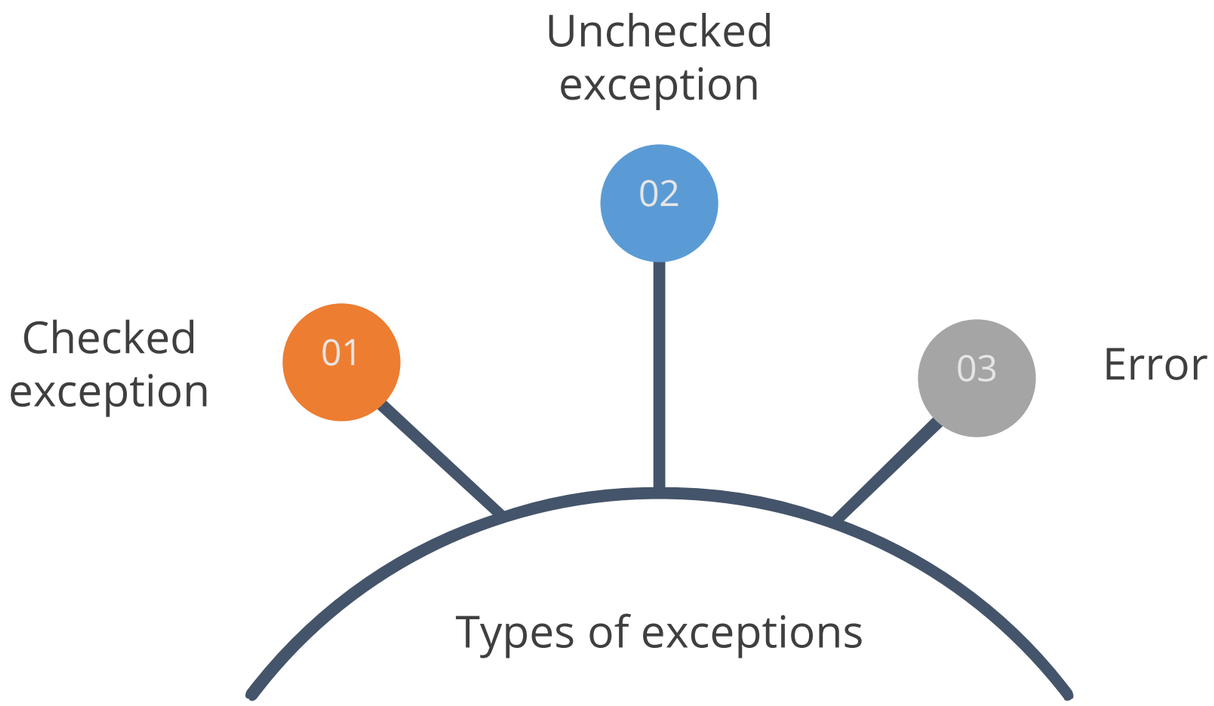
Exception Handling

Hierarchy of Exceptions:



Exception Handling

There are three types of exceptions according to Oracle:



Exception Handling

Types of exceptions:

Checked

The classes that inherit the Throwable class except RuntimeException and Error are called checked exceptions. For example, IOException,etc.

Unchecked

The classes that inherit RuntimeException are called unchecked exceptions. For example, ArrayIndexOutOfBoundsException, etc.

Error

These are irrecoverable. Some examples are OutOfMemoryError, VirtualMachineError, AssertionError, etc.

Exception Handling : Major Keywords

Keywords used in Exception Handling:

Keyword	Description
try	It is a block that contains a set of statements where an exception may occur
catch	The **catch** block handles the uncertain exception in the **try** block.
throw	It is a keyword that manually transfers control from **try** to **catch**.
throws	It is a keyword that handles exceptions without the **try** and **catch**.
finally	The **finally** block contains a set of statements that will always execute whether there is an exception.

Exception Handling

Example of Exception Handling:

```
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);
       }

    System.out.println("The remaining code...");
  }
}
```

Output:

```
java.lang.ArithmeticException: / by zero
The remaining code...
```

Exception Handling

Common Exception Scenarios:

1. ArithmeticException:

ArithmeticException occurs if a number is divided by zero.


```
int a=55/0;//ArithmeticException

float f=66/0; //ArithmeticException
```

Exception Handling

Common Exception Scenarios:

2. NullPointerException:

If a variable has a null value, NullPointerException occurs while performing an operation on the same.

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

Exception Handling

Common Exception Scenarios:

3. ArrayIndexOutOfBoundsException:

ArrayIndexOutOfBoundsException occurs when the array exceeds its size.

```
int sample[]=new int[20];
a[30]=80; //ArrayIndexOutOfBoundsException
```

Exception Handling : try-catch

- The code that may throw an exception is kept in the **try** block.
- Since after the exception occurs in the try block, the inside code does not execute, it is advised not to keep the code which will not throw exception.

Syntax of try-catch block:

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

catch (Exception_class_Name ref) {}
```

Exception Handling : try-catch

Example of try-catch block:

```
public class Sample {

    public static void main(String[] args) {
        try
        {
        int data=50/0;
         System.out.println("The remaining code...");
        }


        catch(ArithmeticException e)
        {
            System.out.println(e);
        }

    }
}
```

Output:

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

Exception Handling : finally

The **finally** block contains a set of statements that will always execute whether there is an exception or not.

```
class FinallyExample{
public static void main(String[] args){
try{
int x=300/0;
}
catch(Exception e)
{System.out.println(e);}

finally{System.out.println("finally block is executed");
}}
```

Output: java.lang.ArithmeticException: / by zero
finally block is executed

Exception Handling

Comparison of **finally**, **Final**, and **finalize** keywords:

final	finally	finalize
final is used to apply restrictions on class, method, and variable.	The finally block is used to place important code. It will be executed whether an exception is handled or not.	The finalize keyword is used to perform clean-up processing just before object is garbage collected.
final is a keyword.	finally is a block.	finalize is a method.

Exception Handling : throw

The “throw” is a keyword that manually transfers control from **try** to **catch**.

```
public class Sample {
    //function to check eligibility to vote
    public static void validate(int age) {
        if(age<18) {

            throw new ArithmeticException(“Not eligible to vote");
        }
        else {
            System.out.println(“Eligible to vote!!");
        }
    }

    public static void main(String args[]){

        validate(11);
        System.out.println(“The remaining code...");
  }
}
```

Output: Exception in thread “main” java.lang.ArithmeticException:
Not eligible to vote

Exception Handling : throws

"throws" is a keyword that handles exceptions without **try** and **catch**.

```
import java.io.IOException;
class Testthrows1
{
  void m()throws IOException{
    throw new IOException("device error");//checked
exception
  }
  void n()throws IOException{
    m();
  }
void p(){
   try{
    n();
 }
```

```
catch(Exception e){System.out.println("exception
corrected");}
  }
  public static void main(String args[]){
   Testthrows1 obj=new Testthrows1();
   obj.p();
   System.out.println("Program works fine...");
  }
}
```

Output: Exception corrected
Program works fine...

Exception Handling

Comparison of **throw** and **throws**:

throw	throws
Java **throw** keyword is used to explicitly throw an exception.	Java **throws** keyword is used to declare an exception.
The checked exception cannot be propagated using **throw** only.	The checked exception can be propagated with **throws**.
It is followed by an instance.	It is followed by class.
It is used within the method.	It is used with the method signature.
It is not possible to throw multiple exceptions.	Multiple exceptions can be declared, for example, public void method()throws IOException and SQLException.

Exception Handling : Custom Exceptions

- Java allows users to create their own exceptions which is a derivation of the “Exception” class.
- These can be called custom exceptions or user-defined exceptions.
- In case of some business logic exceptions where user wants specific handling of the exception custom exceptions are required.

Exception Handling : Custom Exceptions

Example of a custom exception:

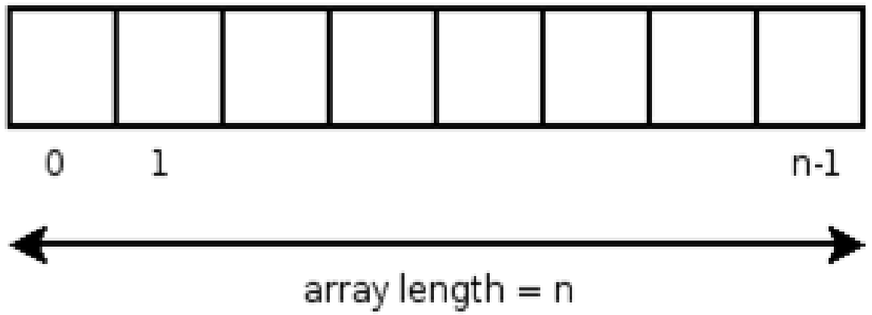
```
class InvalidAgeException extends Exception
{
 InvalidAgeException(String s){
  super(s);
 }
}
class TestCustomException1
{
 static void validate(int age)throws InvalidAgeExce
ption {
  if(age<18)
   throw new InvalidAgeException("not valid");
  else
    System.out.println("welcome to vote");
 }
 public static void main(String args[]){
 try{
  validate(13);
  }catch(Exception m){System.out.println("Exception
 occured:"+m);}     System.out.println("Code is
executing");
 }
}
```

Output:

```
Exception occurred:
InvalidAgeException:not valid
Code is executing
```

Arrays

Arrays : One Dimensional



One-dimensional array

- An array is a data structure that stores elements of similar data types.
- In Java, arrays are index-based, i.e., the first element is stored at the 0^{th} index.

Arrays : One Dimensional

The syntax for declaring a one-dimensional array:

dataType[] arrName; (or)

dataType []arrName; (or)

dataType arrName [];

The syntax for initializing a one-dimensional array:

arrayName=**new** datatype[size];

Arrays : One Dimensional

Example of a declaring, instantiating, and initializing a one-dimensional array:

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

int a[]=new int[5];                    //declaration and instantiation
a[0]=10;                                      //initialization
a[1]=20;
a[2]=60;
a[3]=40;
a[4]=50;

//printing array
for(int i=0;i<a.length;i++)
System.out.println(a[i]);

}}
```

Output:

```
10
20
60
40
50
```

Arrays : One Dimensional

Example of deleting an element:

```
import java.util.Scanner;
public class Delete
{ public static void main(String[] args)
    {   int n, x, flag = 1, loc = 0;
        Scanner s = new Scanner(System.in);
        System.out.print("Enter no. of elements you
want in array:");
        n = s.nextInt();
        int a[] = new int[n];
        System.out.println("Enter all the
elements:");
        for (int i = 0; i < n; i++) {
            a[i] = s.nextInt();
        }
        System.out.print("Enter the element you
want to delete:");
        x = s.nextInt();
        for (int i = 0; i < n; i++)
        {  if(a[i] == x)
            {   flag =1;
                loc = i;
                break;
            }
```

```
else {
                flag = 0;
            }
        }
        if(flag == 1)
        {   for(int i = loc+1; i < n; i++)
            {
                a[i-1] = a[i];
            }
            System.out.print("After Deleting:");
            for (int i = 0; i < n-2; i++)
            {
                System.out.print(a[i]+",");
            }
            System.out.print(a[n-2]);
        }
        else
        {
            System.out.println("Element not
found");
        }
    }
}
```

Arrays : One Dimensional

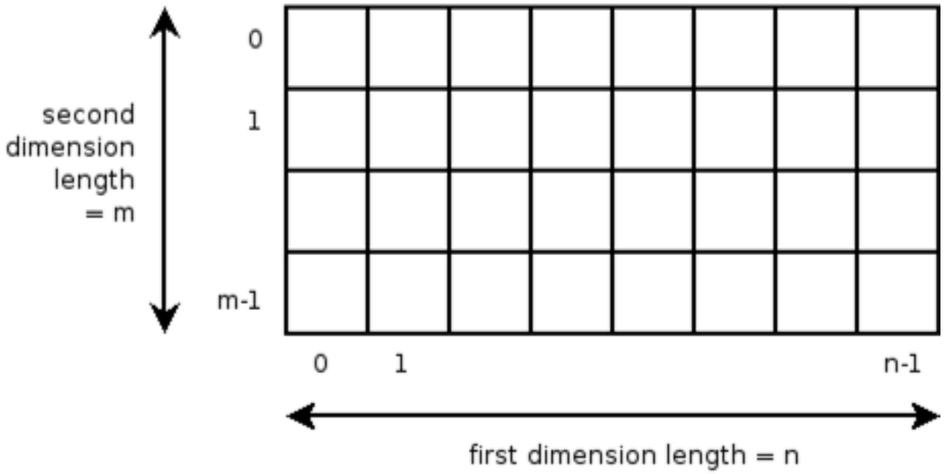
Example of deleting an element (continued):

Output:

```
$ javac Delete.java
$ java Delete
Enter number of elements you want in array: 5
Enter all the elements: 3 5 8 1 4
Enter the element you want to delete: 5
After Deleting: 3 8 1 4
```

Arrays : Multi-Dimensional

In a multi-dimensional array, data is stored in the matrix form, i.e., in row and column index.



The syntax for declaring a multi-dimensional array:

dataType[][] arrName; (or)

dataType [][]arrName; (or)

dataType arrName[][]; (or)

dataType []arrName[];

Arrays : Multi-Dimensional

Example of instantiating a multi-dimensional array:

```
int[][] array1=new int[3][3];//3 row and 3 column
```

Example of initializing a multi-dimensional array:

```
array1[0][0]=1;
array1[0][1]=2;
array1[0][2]=3;
array1[1][0]=4;
array1[1][1]=5;
array1[1][2]=6;
array1[2][0]=7;
array1[2][1]=8;
array1[2][2]=9;
```

Arrays : Multi-Dimensional

Example of a declaring, instantiating, and initializing a two-dimensional array:

```
class Sample{
public static void main(String args[])
{
int arr[][]={{1,2,3},{2,3,4},{3,4,5}};

for(int i=0;i<3;i++){
 for(int j=0;j<3;j++){
   System.out.print(arr[i][j]+" ");
 }
 System.out.println();
}
}}
```

Output:

```
1 2 3
2 3 4
3 4 5
```

Arrays : Multi-Dimensional

Example of a two-dimensional array with a different number of rows and columns:

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


        int arr[][] = new int[3][];
        arr[0] = new int[2];
        arr[1] = new int[3];
        arr[2] = new int[4];
        //initializing a jagged array
        int count = 0;
        for (int i=0; i<arr.length; i++)
            for(int j=0; j<arr[i].length; j++)

                arr[i][j] = count++;
```

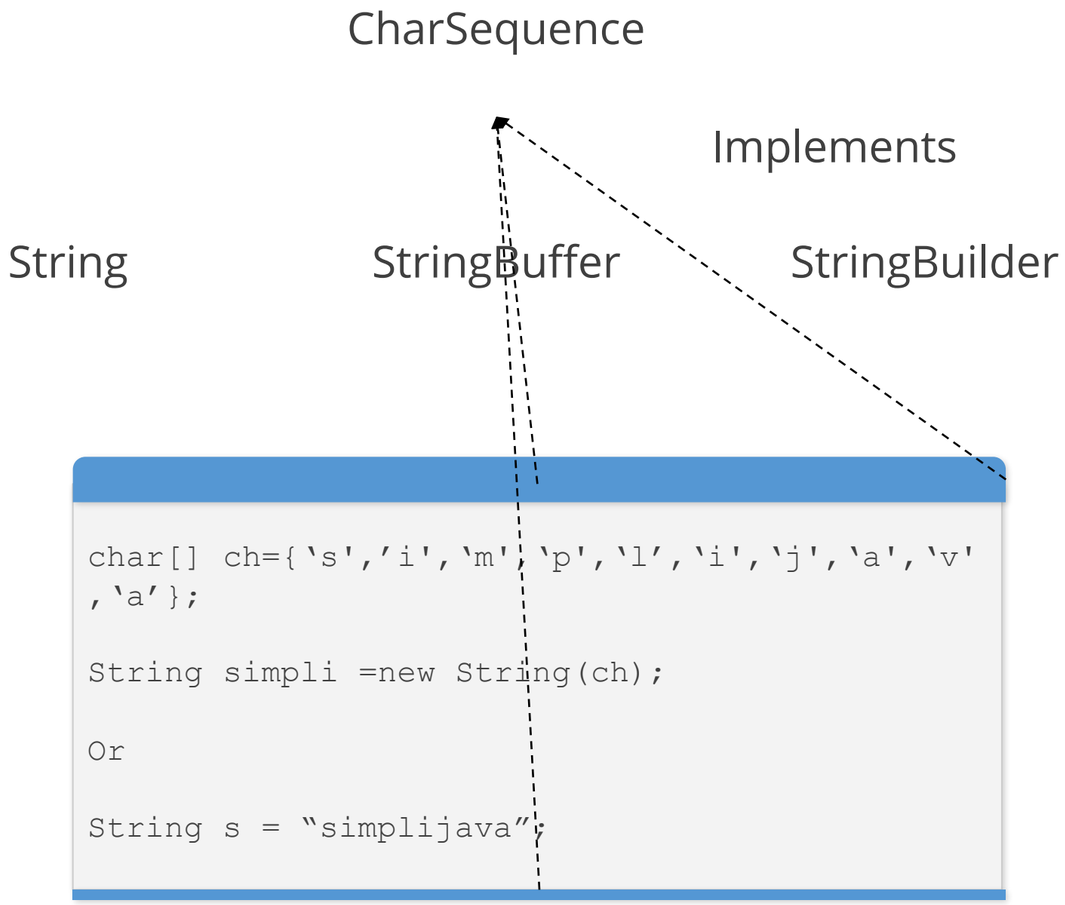
```
for (int i=0; i<arr.length; i++){
            for (int j=0; j<arr[i].length; j++
){
                System.out.print(arr[i][j]+" "
);
            }
            System.out.println();
        }
    }
}
```

Output:

```
01
234
5678
```

Strings

Strings

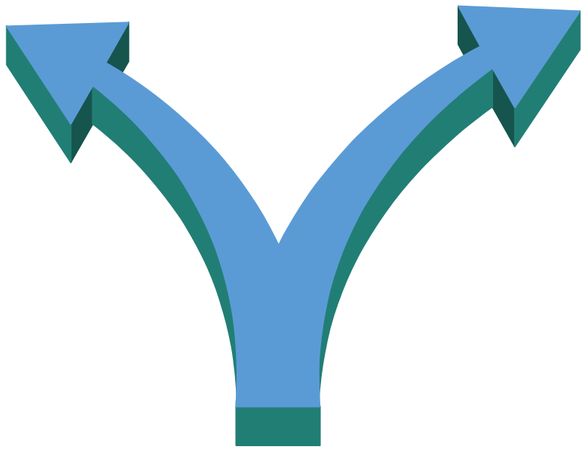


- In Java, an object that represents a sequence of char (character) values is called **String.**
- **String** object in Java is created using the java.lang.String class.
- A **String** object is immutable and cannot be modified.
- Java **String** is equivalent to an array of char values.
- The CharSequence interface is used to represent sequence of characters
- String class, along with StringBuffer and StringBuilder classes implement it,

Strings

There are two ways to create String objects:

By String literal



By **new** keyword

Strings

There are two ways to create String objects:

String literal

- It is created by using double quotes:

  String stringName=“stringValue";

- If the stringValue is not unique, JVM returns the reference to the pooled instance.
- If the stringValue is unique, then a new instance is created in the string constant pool.

```
String simpli1 = “Ola”;
String simpli2 = “Ola”;// doesn’t create a unique instance
```

Strings

There are two ways to create String objects:

new Keyword

- Below is the syntax for creating a String using the “new” keyword:

  String stringVariable = new String(“stringValue");

- In the below example, JVM will create new String object (simpli3) in heap memory and stringValue “Ola” in the string constant pool.

```
instanceString simpli3=new String(“Ola");
```

Strings : Comparison

Strings can be compared using following:

1. equals() method

```
class Test{
 public static void main(String args[]){
   String s1="Rishi";
   String s2="Rishi";
   String s3=new String("Rishi");
   String s4="RISHI";
   System.out.println(s1.equals(s2));//true
   System.out.println(s1.equals(s3));//true
   System.out.println(s1.equals(s4));//false
    System.out.println(s1.equalsIgnoreCase(s4))
;//true

 }
}
```

Output:

```
true
true
false
true
```

Strings : Comparison

2. **==** operator

```
class Test{
 public static void main(String args[]){
   String s1="Rishi";
   String s2="Rishi";
   String s3=new String("Rishi");
    System.out.println(s1==s2);//true  (because
both refer to same instance)
    System.out.println(s1==s3);//false(because
s3 refers to instance created in nonpool)
 }
}
```

Output:
true
false

Strings : Comparison

3. compareTo() method

```
class Test{
 public static void main(String args[]){
   String s1="Ravish";
   String s2="Ravish";
   String s3="Ratan";
   System.out.println(s1.compareTo(s2));//0
   System.out.println(s1.compareTo(s3));//1(be
cause s1>s3)
   System.out.println(s3.compareTo(s1));//-
1(because s3 < s1 )
 }
}
```

Output:

```
0
1
-1
```

Strings : Concatenation

Strings can be concatenated using following:

1. + (String concatenation) operator

```
class Test{
 public static void main(String args[]){
   String s=20+30+"Rishi"+20+20;
   System.out.println(s);//50Rishi2020
 }
}
```

Output:
50Rishi2020

2. concat() method

```
class Test {
 public static void main(String args[]){
   String s1="Rishi ";
   String s2="Raj";
   String s3=s1.concat(s2);
   System.out.println(s3);//Rishi Raj
  }
}
```

Output:
Rishi Raj

Strings : Substring

We can get substrings using any of the following methods:

- publicString substring(int startIndex)
- public String substring(int startIndex, int endIndex)

```
public class Test{
 public static void main(String args[]){
 String r="RishiRaj";
 System.out.println("Original String: " + r);
 System.out.println("Substring starting from index 5: " +r.substring(5));//Raj
 System.out.println("Substring starting from index 0 to 6: "+r.substring(0,5)); //Rishi
 }
}
```

Output:

```
Raj
Rishi
```

Strings : String Class Methods

Some important String class methods:

toUpperCase()	charAt()
toLowerCase()	length()
trim()	intern()
startsWith()	valueOf()
endsWith()	replace()

Strings : String Class Methods

Some important String class methods: Example 1

```
public class Test
{
public static void main(String ar[])
{
String s="Rishi"; s1=" Rishi "
System.out.println(s.toUpperCase());//RISHI
System.out.println(s.toLowerCase());//rishi
System.out.println(s1.trim());//Rishi
System.out.println(s.startsWith("Ri"));//true
 System.out.println(s.endsWith("i"));//true
System.out.println(s.charAt(0));//R
System.out.println(s.charAt(3));//h

}
```

Output:

```
RISHI
rishi
Rishi
true
true
R
h
```

Strings : String Class Methods

Some important String class methods: Example 2

```
public class Stringoperation1
{
public static void main(String ar[])
{
String s="Rishi"; s1=" Rishi "
System.out.println(s.length());//5
String s2=s.intern();
System.out.println(s2);//Rishi

int a=20;
String s3=String.valueOf(a);
System.out.println(s3+20);

String s1="Java is robust, Java is OOPs based language";
String replaceString=s1.replace("Java","Mava");//replaces all occurrences of "Java" to "Mava"
System.out.println(replaceString);

}
```

Output:

```
5
Rishi
2020
Java is robust, Java is OOPs based language
```

Strings : StringBuffer

- StringBuffer class helps in creating mutable (modifiable) String objects.
- At any point in time, it contains a particular sequence of characters, but the length and content of the sequence can be changed through certain method calls.
- Some commonly used methods are append(), insert(), reverse(), replace(int startIndex, int endIndex, String str), delete(int startIndex, int endIndex), capacity(), etc.

Strings : StringBuffer

Example of StringBuffer class methods:

```
public class Test
{
public static void main(String ar[])
{
StringBuffer sb=new StringBuffer("Hello ");
sb.append("Learner");//now original string is changed
System.out.println(sb);//prints Hello Learner

StringBuffer sb1=new StringBuffer("Hello ");
sb1.insert(1,"Java");//now original string is changed
System.out.println(sb1);//prints HJavaello

StringBuffer sb2=new StringBuffer("Hello ");
sb2.replace(1,3,"Java");
System.out.println(sb2);//prints HJavalo

StringBuffer sb3=new StringBuffer("Hello ");
sb3.delete(1,3);
System.out.println(sb3);//prints Hlo

}
```

Output:

```
Hello Learner
Hjavaello
HJavalo
Hlo
```

Strings : StringBuilder

- StringBuilder class helps in creating mutable (modifiable) String objects.
- It is the same as the StringBuffer class, except it is non-synchronized.
- Some commonly used methods are append(), insert(), reverse(), replace(int startIndex, int endIndex, String str), delete(int startIndex, int endIndex), capacity(), etc.

Strings : StringBuilder

Example of StringBuilder class methods:

```
public class Test
{
public static void main(String ar[])
{
StringBuilder sb=new StringBuffer("Hello ");
sb.append("Learner");//now original string is changed
System.out.println(sb);//prints Hello Learner

StringBuilder sb1=new StringBuffer("Hello ");
sb1.insert(1,"Java");//now original string is changed
System.out.println(sb1);//prints HJavaello

StringBuilder sb2=new StringBuffer("Hello ");
sb2.replace(1,3,"Java");
System.out.println(sb2);//prints HJavalo

StringBuilder sb3=new StringBuffer("Hello ");
sb3.delete(1,3);
System.out.println(sb3);//prints Hlo

}
```

Output:

```
Hello Learner
Hjavaello
HJavalo
Hlo
```

Strings : StringBuffer vs. StringBuilder

StringBuffer	StringBuider
It is thread safe (synchronized), i.e., two threads cannot call its methods simultaneously.	It is not thread safe (unsynchronized), i.e., two threads can call its methods simultaneously.
StringBuffer is less efficient,	StringBuilder is more efficient.
It was introduced in Java 1.0.	It was introduced in Java 1.5.

Inner Class

Inner Class

- Inner class is a class declared inside the class or interface that can access all the members of the outer class, including private data members and methods.
- Syntax:

```
class Outer_class
{   //code statements
        class Inner_class{      //code statements
        }
}
```

Inner Class

Advantages of inner class:

- Inner class is accessible by all data members and methods of the outer class, including private data members and methods.
- It develops more maintainable and readable code.
- It requires less coding effort.
- When the access for any other class needs to be restricted, it is advised to program an inner class.

Multithreading

Multithreading : Thread

- A thread is an independent path of execution within a program.
- The *java.lang.Thread class* enables you to create and control threads.

A thread is composed of three main parts:

- Virtual CPU
- The code that the CPU executes
- The data on which the code works

Multithreading : Creating a Thread

- A thread can be created by “Thread” class or by implementing the Runnable interface.
- Commonly used constructors of “Thread” class:

  Thread()

  Thread(String name)

  Thread(Runnable r)

  Thread(Runnable r, String name)

Multithreading : Creating a Thread

To create a thread, java.lang.Runnable is preferred over java.lang.Thread.

- As Java doesn't support multiple inheritance, the extended Thread class will lose the chance to further extend or inherit other classes.
- Implementing the Runnable interface will make the code easily maintainable as it does a logical separation of tasks from the runner.
- In object-oriented programming, extending a class means modifying or improving the existing class. Implementing Runnable is a good practice.

Multithreading : Creating a Thread

Example of creating a thread by extending the Thread class:

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

Output:
Thread is running

Multithreading : Creating a Thread

Example of creating a thread by implementing the Runnable interface:

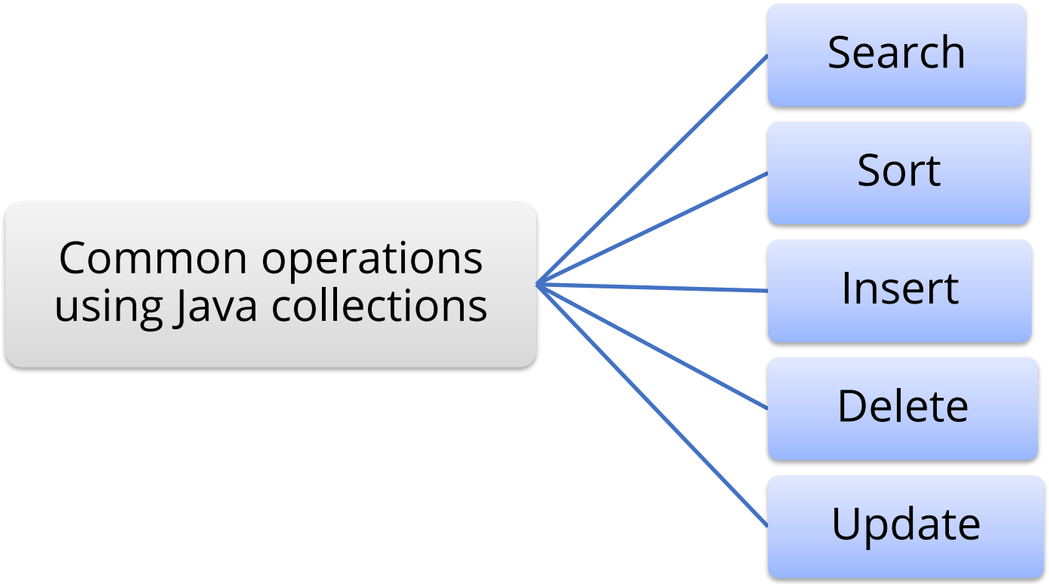
```
class Test implements Runnable{
public void run(){
System.out.println("Thread is running");
}

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

Output:

```
Thread is running
```

Collections

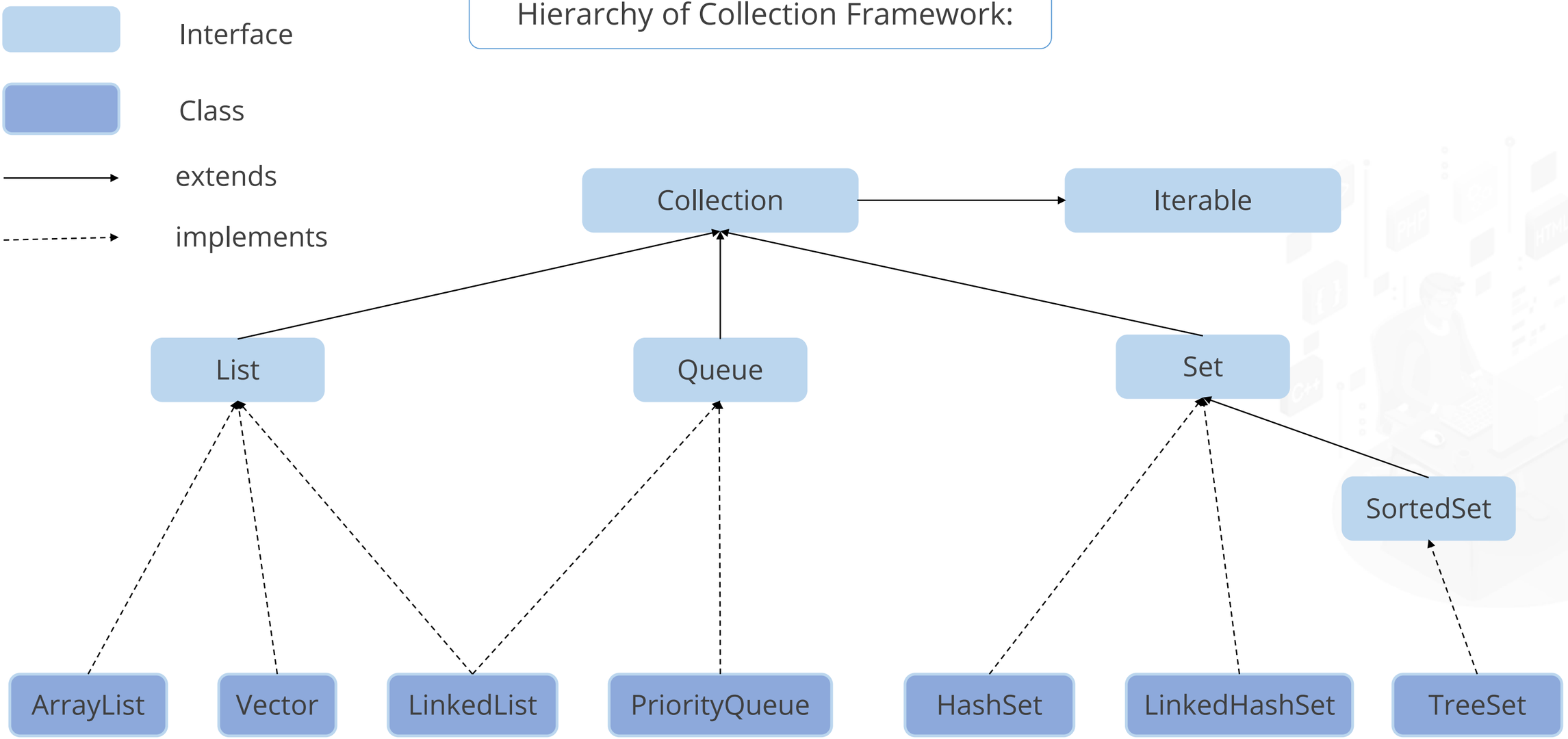
Collections



- A collection is a group or single unit of individual objects.
- Java Collection Framework defines several classes and interfaces to represent objects as a single unit.

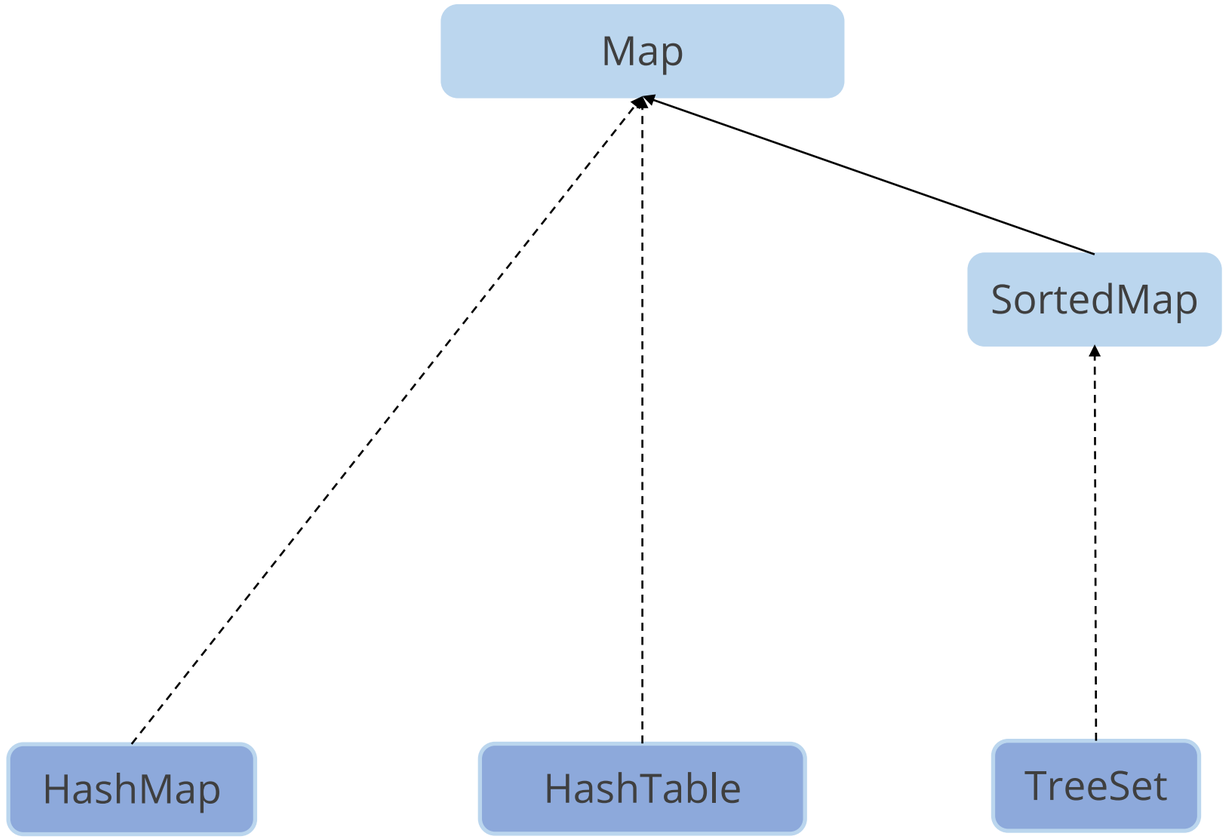
Collections : Hierarchy

Hierarchy of Collection Framework:



Collections : Hierarchy

Hierarchy of Map:



Collections

Legacy Classes vs. Collection Framework

Legacy Classes	Collection Framework
Legacy classes comprise classes and interfaces supported by older versions of Java.	Collections are used to store and manipulate a group of objects.
Legacy classes include Dictionary, Hashtable, Properties, Stack, and Vector.	It includes interfaces like Collection, Set, List, or Map. Classes include ArrayList, LinkedList, Vector and Stack.
Legacy classes are synchronized.	Collections are non-synchronized.

Collections

Child interfaces of List interface:

List

- An interface that contains sequentially arranged elements
- ArrayList, LinkedList, and Vector subclasses implement the List interface

Set

- Contains unique elements arranged in any order
- HashSet, LinkedHashSet, TreeSet classes implement the Set interface

Queue

- Contains an ordered list of homogeneous elements, where elements are added and removed at the rear and front end respectively
- LinkedList, Priority queue, and ArrayQueue implement this interface

Collections : LinkedList

It contains elements of a specified collection, arranged into an ordered list.

```
import java.util.*;
public class Test {
public static void main(String args[]){
LinkedList<String> al=new LinkedList<String>();
al.add("Kavi");
al.add("Vijay");
al.add("Ravi");
al.add("Ajay");
Iterator<String> itr=al.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

```
Output:
Kavi
Vijay
Ravi
Ajay
```

Collections : Methods

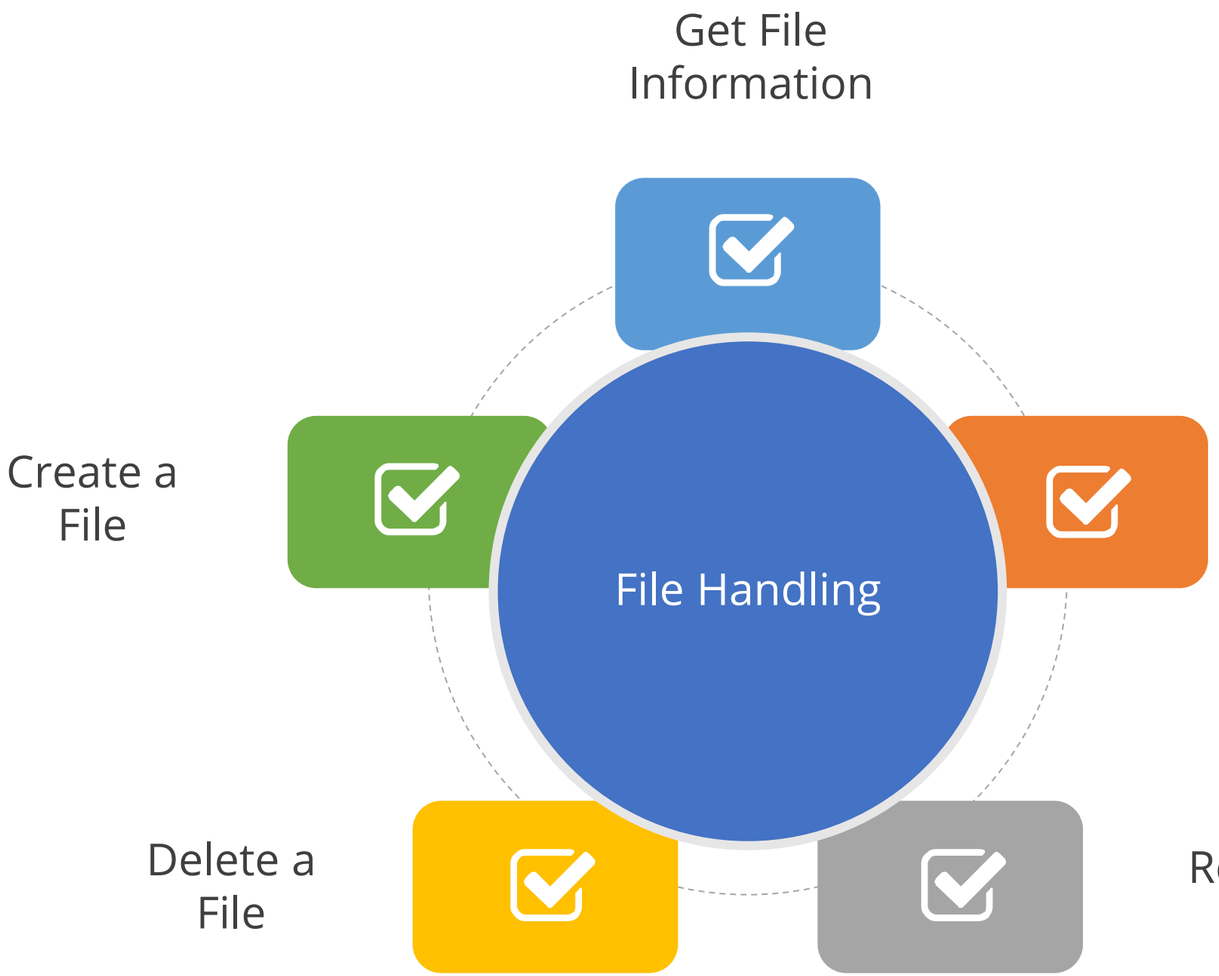
Some commonly used collection methods:

size()	clear()
add(Object o)	isEmpty()
addAll(Collection c)	iterator()
remove(Object o)	equals(Object o)
removeAll(Collection c)	toArray()
contains(Object o)	toArray(Object[] a)
containsAll(Collection c)	retainAll(Collection c)

File Handling and Serialization

File Handling

We can perform the following operations on a file:



File Handling : Create a File

To create a file, createNewFile() method is used.

```
import java.io.File;   // Importing File class
// Importing the IOException class for handling errors
import java.io.IOException;
 class CreateFile {

        public static void main(String args[]) {

         try {  // Creating an object of a file
         File f1 = new File("D:Sample.txt");
         if (f1.createNewFile()) {
              System.out.println("File " + f1.getName() + " is
 created successfully.");         }
      else {
      System.out.println("File already exists in directory.");
       }
     } catch (IOException exception)
       {
        System.out.println("An unexpected error occurred.");
              exception.printStackTrace();
}   }  }
```


Output:
File Sample.txt is
created successfully

File Handling : Get File Information

The methods like name, absolute path, is readable, etc. can be used to get file information.

```
import java.io.File;
class FileInfo {
    public static void main(String[] args) {
        // Creating file object
        File f0 = new File("D:Sample.txt");
        if (f0.exists()) {
            // Getting file name
            System.out.println("The name of the fil
e is: " + f0.getName());

            // Getting path of the file
            System.out.println("The absolute path o
f the file is: " + f0.getAbsolutePath());

            // Checking whether the file is writabl
e or not
            System.out.println("Is file writeable?:
 " + f0.canWrite());
```

```
// Checking whether the file is readable or not
            System.out.println("Is file readable "
+ f0.canRead());

// Getting the length of the file in bytes
            System.out.println("The size of the fil
e in bytes is: " + f0.length());
        }
else {
   System.out.println("The file does not exist.");

        }
    }
}
```

```
Output:
The name of the file is: Sample.txt
The absolute path of the file is
D:\\Sample.txt
Is file writeable?: true
Is file readable true
The size of the file in bytes is: 0
```

File Handling : Write to a File

To write data into a file, the write() method of the FileWriter class is used here.

```
import java.io.FileWriter;
import java.io.IOException;

class WriteToFile {
    public static void main(String[] args) {

    try{
       FileWriter fwrite = new FileWriter("D:Sample
.txt");
        // writing the content into the file
          fwrite.write("File Handling is being done
for this File.");
// Closing the stream
        fwrite.close();
        System.out.println("Content is successfully
 written to the file.");
    }
```

```
 catch (IOException e) {
        System.out.println("Unexpected error occurr
ed");
        e.printStackTrace();
        }
    }
}
```

```
Output:
Content is successfully written to the file.
```

File Handling : Read from a File

To read data from a file, the Scanner class is used here.

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

class ReadFromFile {
    public static void main(String[] args) {
        try {
            // Create f1 object of the file to read
 data
            File f1 = new File("D:Sample.txt");
        Scanner dataReader = new Scanner(f1);
    while (dataReader.hasNextLine()) {
        String fileData = dataReader.nextLine();
                System.out.println(fileData);
            }
```

```
dataReader.close();
        } catch (FileNotFoundException exception) {

            System.out.println("Unexcpected error o
ccurred!");
            exception.printStackTrace();
        }
    }
}
```

```
Output:
File Handling is being done for this File.
```

File Handling : Delete a File

To delete a file, the delete() method is used here.

```
import java.io.File;
class DeleteFile {
  public static void main(String[] args) {
    File f0 = new File("D:Sample.txt");
    if (f0.delete()) {
      System.out.println(f0.getName()+ " file is de
leted successfully.");
    } else {
      System.out.println("Unexpected error found in
 deletion of the file.");
    }
  }
}
```

```
Output:
Sample.txt file is being is deleted
successfully.
```

Serialization

- Serialization is a mechanism for saving the objects as a sequence of bytes and rebuilding the byte sequence back into a copy of the object later.
- For a class to be serialized, the class must implement the java.io.Serializable interface.
- The Serializable interface has no methods. It only serves as a marker, which indicates that the class that implements the interface can be considered for serialization.

Serialization

Example of serialization:

```
import java.io.*;
public class Student implements Serializable{
 int id;
 String name;
 public Student(int id, String name) {
  this.id = id;
  this.name = name;
 }
}

class Persist{
 public static void main(String args[]){
  try{
  //Creating the object
  Student s1 =new Student(211,"Avi");
```

```
//Creating stream and writing the object
  FileOutputStream fout=new FileOutputStream("f.txt
");
  ObjectOutputStream out=new ObjectOutputStream(fou
t);
  out.writeObject(s1);
  out.flush();
  //closing the stream
  out.close();
  System.out.println("Success");
  }catch(Exception e){System.out.println(e);}
 }
}
```

```
Output:
Success
```

Serialization

Example of deserialization:

```
import java.io.*;

public class Student implements Serializable{
 int id;
 String name;
 public Student(int id, String name) {
  this.id = id;
  this.name = name;
 }
}

class Depersist{
 public static void main(String args[]){
  try{
  //Creating stream to read the object
  ObjectInputStream in=new ObjectInputStream(new Fi
leInputStream("f.txt"));
```

```
  Student s=(Student)in.readObject();
  //printing the data of the serialized object
  System.out.println(s.id+" "+s.name);
  //closing the stream
  in.close();
  }catch(Exception e){System.out.println(e);}
 }
}
```

```
Output:
211 Avi
```

Garbage Collection

Garbage Collection

Garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects.

Advantages of garbage collection:

- It clears off unreferenced objects from heap memory, which makes Java memory efficient.
- The garbage collector, which is a component of the JVM, performs it automatically, so there is no need for additional effort.

Garbage Collection

Some of the ways to un-refer an object are mentioned below:

```
Student s = new Student();
s=null;         // null by reference

************

Student s1 = new Student();
Student s2 = new Student();   //reference to others

************

new Student();          // anonymous object
```

Garbage Collection

- The finalize() method is invoked each time before the object is garbage collected.
- The gc() method can be used to invoke the garbage collector as it performs the cleanup.

```
public class Test{

public void finalize(){
System.out.println("Garbage got collected");
}

public static void main(String args[]){
  Test1 s1=new Test();
  Test1 s2=new Test ();
  s1=null;
  s2=null;
  System.gc();
 }
}
```

```
Output:
Garbage got collected
Garbage got collected
```

Key Takeaways

- Java is a platform-independent programming language and computing platform for developing application software that can run on any device.
- Java implements all the OOPs principles, which makes it a robust and secure language.
- The Java platform differs from most other platforms as it is a software-based platform that runs on top of other hardware-based platforms.
- Java supports dynamic compilation and automatic memory management (garbage collection).