# Module 2

## Introducing Classes, Methods

Introducing Classes: Class Fundamentals, Declaring Objects, Assigning Object Reference Variables, Introducing Methods, Constructors, The this Keyword, Garbage Collection. Methods and Classes: Overloading Methods, Objects as Parameters, Argument Passing, Returning Objects, Recursion, Access Control, Understanding static, Introducing final, Introducing Nested and Inner Classes..

# Classes

Classes fundamentals; Declaring objects; Constructors, this keyword, garbage collection.

## Object

- Any entity that has state and behavior is known as an object. For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical.
- An Object can be defined as an instance of a class.
- An object contains an address and takes up some space in memory.
- Objects can communicate without knowing the details of each other's data or code.
- The only necessary thing is the type of message accepted and the type of response returned by the objects.
- **Example:** A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.

# Classes Fundamentals

Classes fundamentals; Declaring objects; Assigning object Reference variables

## Class

- *Collection of objects* is called class. It is a logical entity.
- A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.

## Inheritance

- *When one object acquires all the properties and behaviors of a parent object, it is known as inheritance.*
- *It provides code reusability.*
- *It is used to achieve runtime polymorphism.*

# Classes

Classes fundamentals; Declaring objects; Constructors, this keyword, garbage collection.

## Polymorphism

- *If one task is performed in different ways, it is known as polymorphism.*
- *For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.*
- *Another example can be to speak something; for example, a cat speaks meow, dog barks woof, etc.*
- *In Java, we use method overloading and method overriding to achieve polymorphism.*

# Classes

Classes fundamentals; Declaring objects; Constructors, this keyword, garbage collection.

## Abstraction

- *Hiding internal details and showing functionality is known as abstraction.*
- *For example phone call, we don't know the internal processing.*
- *In Java, we use abstract class and interface to achieve abstraction.*

## Encapsulation

- *Binding (or wrapping) code and data together into a single unit are known as encapsulation.*
- *For example, a capsule, it is wrapped with different medicines.*
- *A java class is the example of encapsulation.*
- *Java bean is the fully encapsulated class because all the data members are private here.*

Capsule

# Classes

## Advantage of OOPs over Procedure-oriented programming language

1) *OOPs makes development and maintenance easier, whereas, in a procedure-oriented programming language, it is not easy to manage if code grows as project size increases.*
2) *OOPs provides data hiding, whereas, in a procedure-oriented programming language, global data can be accessed from anywhere.*
3) *OOPs provides the ability to simulate real-world event much more effectively. We can provide the solution of real word problem if we are using the Object-Oriented Programming language.*

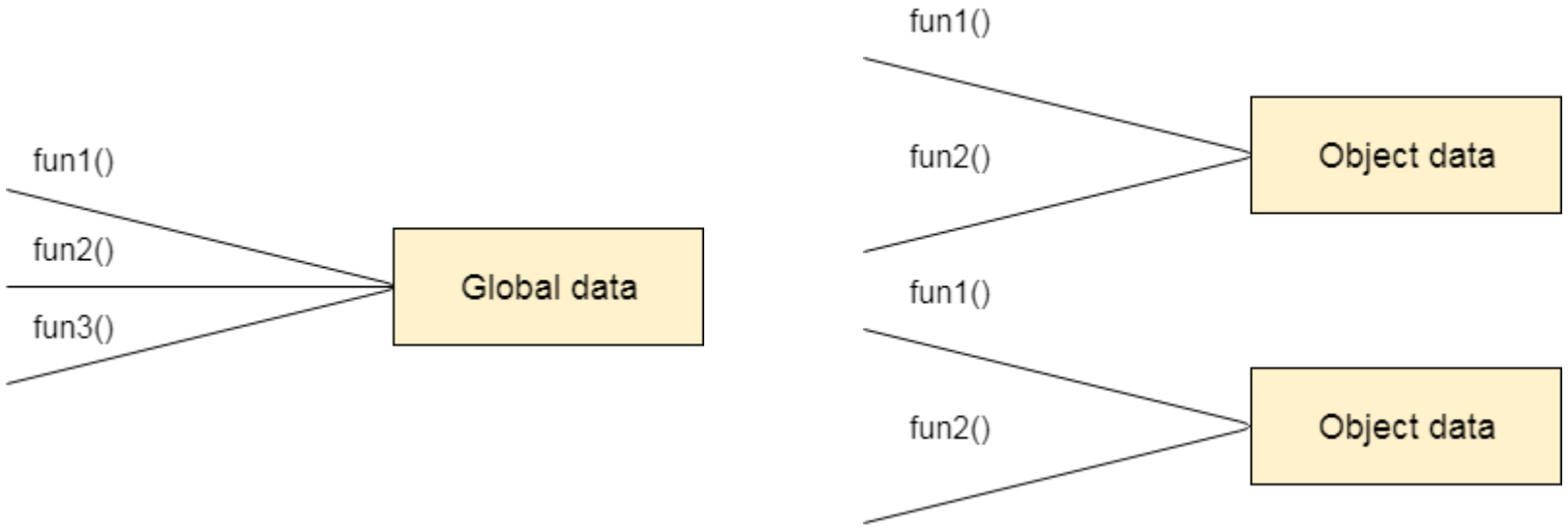
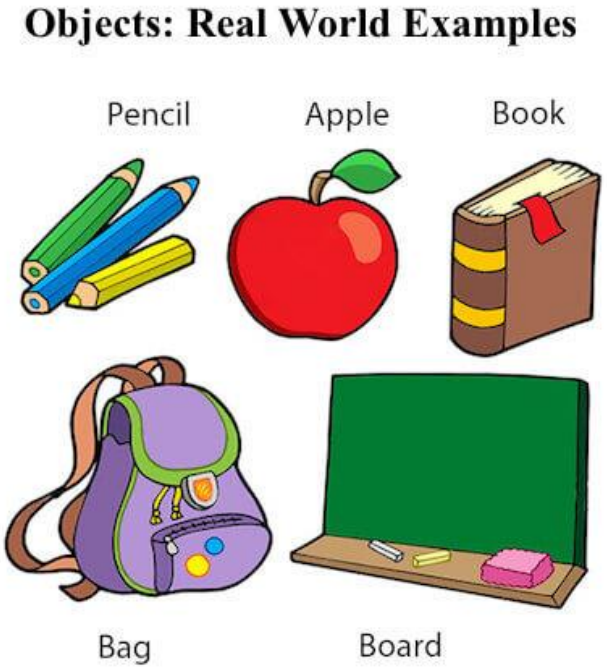



Figure: Data Representation in Procedure-Oriented Programming

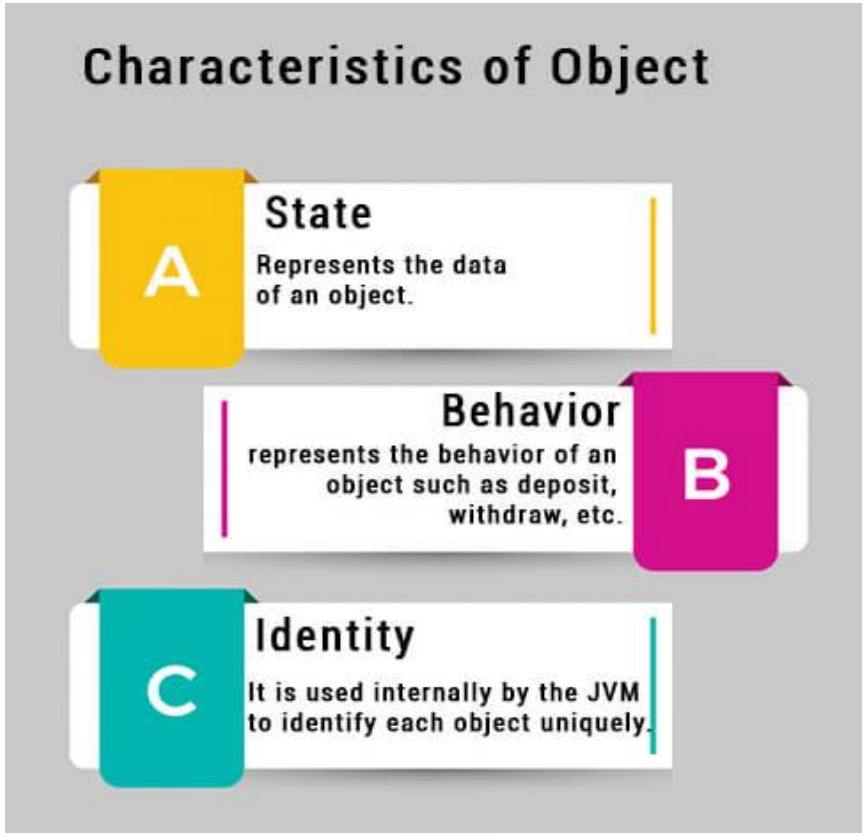
# Objects and Classes in Java

- An **object** in Java is the physical as well as a logical entity, whereas, a **class** in Java is a logical entity only.
- An entity that has state and behavior is known as an **object** e.g., chair, bike, marker, pen, table, car, etc.
- It can be physical or logical (tangible and intangible).
- The example of an intangible object is the banking system.



# Characteristics of Objects

- **State:** represents the data (value) of an object.
- **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.
- **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.



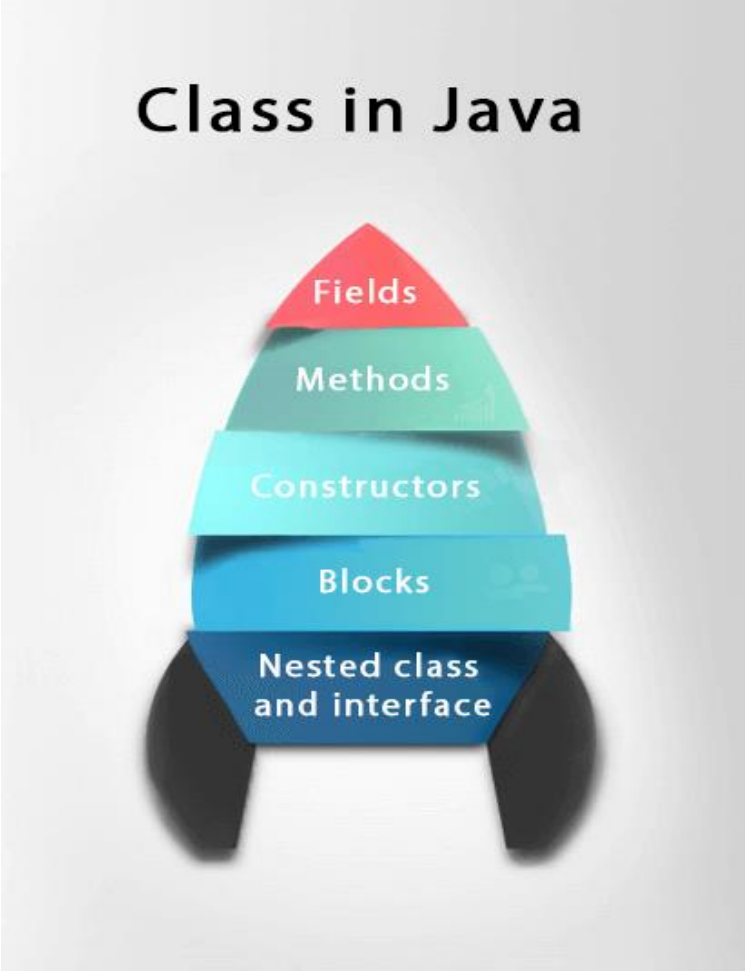

- For Example, Pen is an object.
- Its name is Reynolds; color is white, known as its state.
- It is used to write, so writing is its behavior.

# Class

- A class is a group of objects which have common properties.
- It is a template or blueprint from which objects are created. It is a logical entity.
- It can't be physical.

A class in Java can contain:

**Fields**
**Methods**
**Constructors**
**Blocks**
**Nested class and interface**




Syntax to declare a class:

```
class <class_name>{
    field;
    method;
}
```

# Objects

**An object is an instance of a class.**

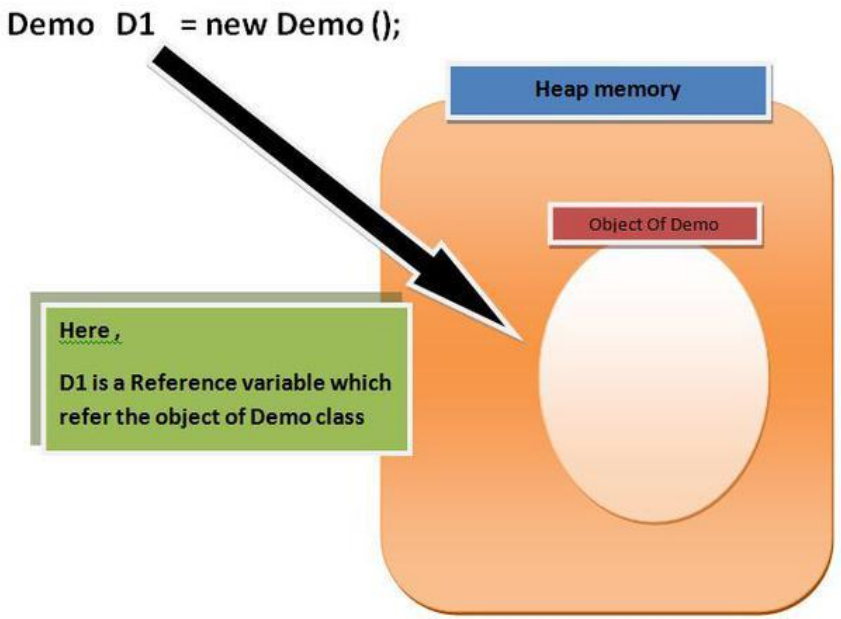
- A class is a template or blueprint from which objects are created.
- So, an object is the instance(result) of a class.

**Object Definitions:**

- An object is *a real-world entity*.
- An object is *a runtime entity*.
- The object is *an entity which has state and behavior*.
- The object is *an instance of a class*.

# Assigning Objects Reference variable

- In Java, objects are created dynamically on the heap memory, and reference variables are used to hold the memory address of these objects
- . This concept of reference variables is fundamental to Java's approach to object-oriented programming.



# Assigning Objects Reference variable

- **Reference variable is a variable that holds the memory address of an object rather than the actual object itself.**
- **It acts as a reference to the object and allows manipulation of its data and methods.**
- **Reference variables are declared with a specific type, which determines the methods and fields that can be accessed through that variable**
- **When an object is created using the new keyword, memory is allocated on the heap to store the object's data.**
- **The reference variable is then used to refer to this memory location, making it possible to access and manipulate the object's properties and behaviours**

```
import java.io.*;
Class Demo {
    int x = 10;
    int display()
    {
        System.out.println("x = " + x);
        return 0;
    }
}

class Main {
    public static void main(String[] args)
    {
        Demo D1 = new Demo(); //point1

        System.out.println(D1); //point 2

        System.out.println(D1.display()); //point 3
    }
```

```
Output

Demo@214c265e

x = 10

0
```

# Introducing Method

A method is like a function which is used to expose the behavior of an object.

Advantage of Method

- Code Reusability
- Code Optimization

new keyword in Java

The new keyword is used to allocate memory at runtime. All objects get memory in Heap memory area.

# Object and Class Example: main within the class

```
//Java Program to illustrate how to define a class and fields
//Defining a Student class.
class Student{
    //defining fields
   int id;//field or data member or instance variable
   String name;
  //creating main method inside the Student class
  public static void main(String args[]){
      //Creating an object or instance
      Student s1=new Student();//creating an object of Student
      //Printing values of the object
  System.out.println(s1.id);//accessing member through reference variable
  System.out.println(s1.name);
 }
}
```

**Output:**
0
null

# Object and Class Example: main outside the class

```
//Java Program to demonstrate having the main method in
//another class
//Creating Student class.
class Student{
 int id;
 String name;
}
//Creating another class TestStudent1 which contains the main method
class TestStudent1{
 public static void main(String args[]){
  Student s1=new Student();
  System.out.println(s1.id);
  System.out.println(s1.name);
 }
}
```

**Output:**
0
null

# Initialize Object

## 3 Ways to initialize object

There are 3 ways to initialize object in Java.

1. By reference variable
2. By method
3. By constructor

# Initialize Object

Initialization through reference

- Initializing an object means storing data into the object.

```
class Student{
 int id;
 String name;
}
class TestStudent2{
 public static void main(String args[]){
  Student s1=new Student();
  s1.id=101;
  s1.name="Sonoo";
  System.out.println(s1.id+" "+s1.name);//printing members with a white space
 }
}
```

**Output:**

101 Sonoo

# Initialize Object

- Creating multiple objects and store information in it through reference variable.

```
class Student{
 int id;
 String name;
}
class TestStudent3{
 public static void main(String args[]){
  //Creating objects
  Student s1=new Student();
  Student s2=new Student();
  //Initializing objects
  s1.id=101;
  s1.name="Sonoo";
  s2.id=102;
  s2.name="Amit";
  //Printing data
  System.out.println(s1.id+" "+s1.name);
  System.out.println(s2.id+" "+s2.name);
 }
}
```

**Output:**

101 Sonoo
102 Amit

# Initialize Object



## Initialization through method

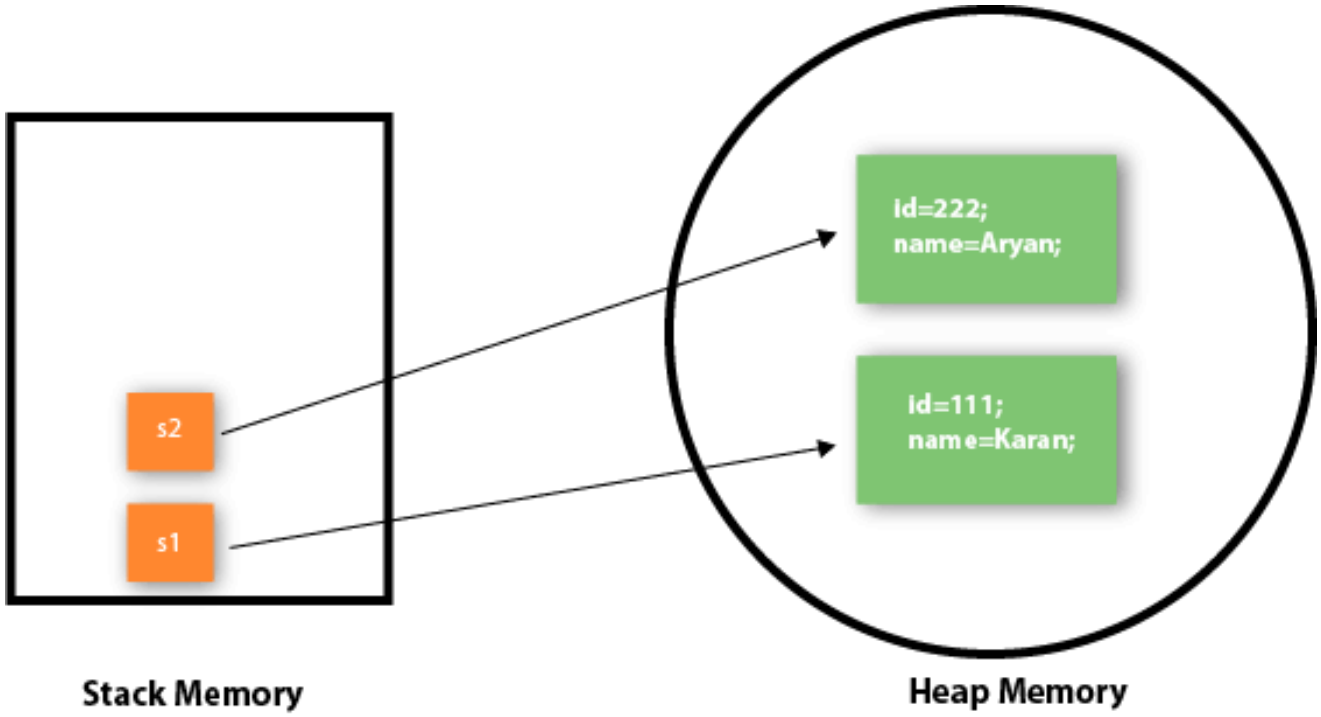
Create the objects of a class and initializing the value to these objects by invoking the methods..

```
class Student{
 int rollno;
 String name;
 void insertRecord(int r, String n){
  rollno=r;
  name=n;
 }
 void displayInformation(){System.out.println(rollno+" "+name);}
}
class TestStudent4{
 public static void main(String args[]){
  Student s1=new Student();
  Student s2=new Student();
  s1.insertRecord(111,"Karan");
  s2.insertRecord(222,"Aryan");
  s1.displayInformation();
  s2.displayInformation();
 }
}
```

**Output:**

111 Karan
222 Aryan

- Object gets the memory in heap memory area.
- The reference variable refers to the object allocated in the heap memory area.
- Here, s1 and s2 both are reference variables that refer to the objects allocated in memory.



# Example

Object and Class Example: Employee

```
class Employee{
   int id;
   String name;
   float salary;
   void insert(int i, String n, float s) {
      id=i;
      name=n;
      salary=s;
   }
   void display(){
system.out.println
(id+" "+name+" "+salary);}
}
```

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

   Employee e1=new Employee();
   Employee e2=new Employee();
   Employee e3=new Employee();
   e1.insert(101,"ajeet",45000);
   e2.insert(102,"irfan",25000);
   e3.insert(103,"nakul",55000);
   e1.display();
   e2.display();
   e3.display();
}
}
```

**Output:**

101 ajeet 45000.0
102 Irfan 25000.0
103 nakul 55000.0

# Object and Class Example: Rectangle

Object and Class Example: Employee

```
class Rectangle{
 int length;
 int width;
 void insert(int l, int w){
  length=l;
  width=w;
 }
 void calculateArea()
{System.out.println(length*
width);}
}
```

```
class TestRectangle1{
public static void main(String args[]){

  Rectangle r1=new Rectangle();
  Rectangle r2=new Rectangle();
  r1.insert(11,5);
  r2.insert(3,15);
  r1.calculateArea();
  r2.calculateArea();
 }
 }
```

**Output:**
**55**
**45**

# Constructors in Java

Constructors, this keyword, garbage collection

- In Java, a constructor is a block of codes similar to the method.
- It is called when an instance of the class is created.
- At the time of calling constructor, memory for the object is allocated in the memory.
- It is a special type of method which is used to initialize the object.
- Every time an object is created using the new() keyword, at least one constructor is called.
- It calls a default constructor if there is no constructor available in the class.
- In such case, Java compiler provides a default constructor by default.

There are two types of constructors in Java:

1. no-arg constructor
2. parameterized constructor.

**Note:** It is called constructor because it constructs the values at the time of object creation. It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.

# Rules for creating Java constructor

There are two rules defined for the constructor.

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized.

**Abstraction** is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

# Rules for creating Java constructor

**Static:** The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc. The static variable gets memory only once in the class area at the time of class loading. TestStaticVariable1.java

**Final:** If you make any variable as final, you cannot change the value of final variable (It will be constant).

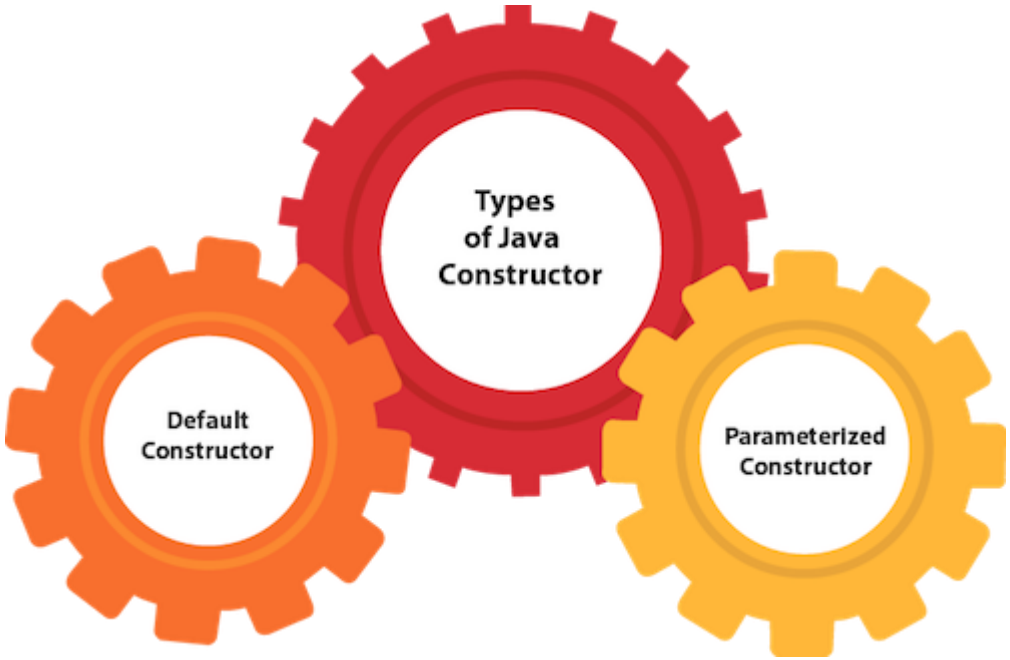
## Example of final variable

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed. Bike9.java

# Types of Java constructors

There are two types of constructors in Java:

- Default constructor (no-arg constructor)
- Parameterized constructor



# Java Default Constructor

A constructor is called "Default Constructor" when it doesn't have any parameter.

Syntax of default constructor:

<class_name>(){}

## Example of default constructor

```
//Java Program to create and call a default constructor
class Bike1{
//creating a default constructor
Bike1(){System.out.println("Bike is created");}
//main method
public static void main(String args[]){
//calling a default constructor
Bike1 b=new Bike1();
}
}
```

# Java Default Constructor

In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation.


Q) What is the purpose of a default constructor?

The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

Example of default constructor that displays the default values

**Explanation:**In the above class, you are not creating any constructor so compiler provides you a default constructor. Here 0 and null values are provided by default constructor.

# Java Parameterized Constructor

- A constructor which has a specific number of parameters is called a parameterized constructor.

Use of the parameterized constructor

- The parameterized constructor is used to provide different values to distinct objects.
- However, you can provide the same values also.

Example of parameterized constructor

# Constructor Overloading in Java

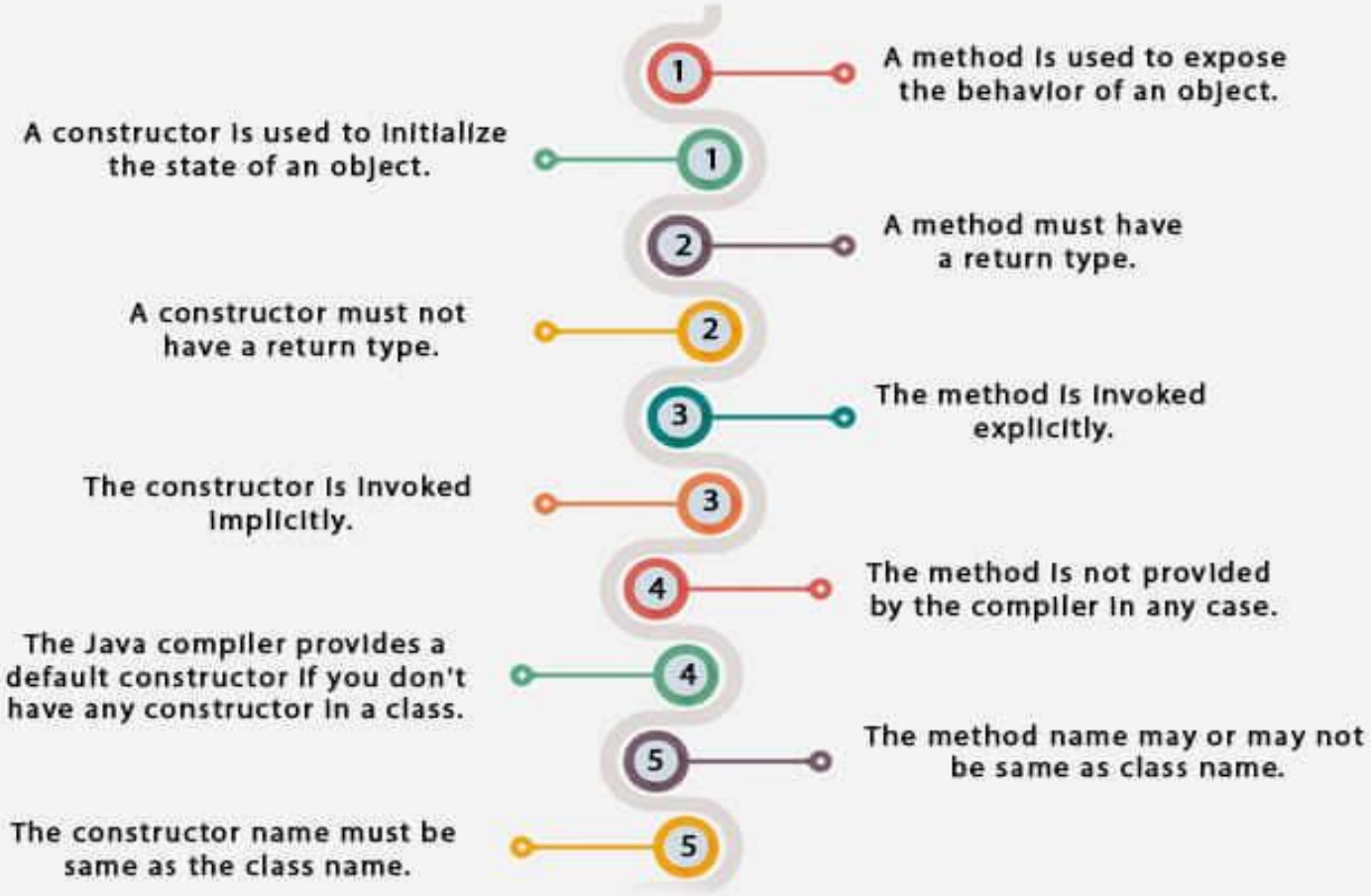
- In Java, a constructor is just like a method but without return type.
- It can also be overloaded like Java methods.
- Constructor overloading in Java is a technique of having more than one constructor with different parameter lists.
- They are arranged in a way that each constructor performs a different task.
- They are differentiated by the compiler by the number of parameters in the list and their types.

Example of Constructor Overloading

# Difference between constructor and method in Java

| Java Constructor | Java Method |
|---|---|
| A constructor is used to initialize the state of an object. | A method is used to expose the behavior of an object. |
| A constructor must not have a return type. | A method must have a return type. |
| The constructor is invoked implicitly. | The method is invoked explicitly. |
| The Java compiler provides a default constructor if you don't have any constructor in a class. | The method is not provided by the compiler in any case. |
| The constructor name must be same as the class name. | The method name may or may not be same as the class name. |

# Difference between constructor and method in Java



# Java Copy Constructor

- There is no copy constructor in Java.
- However, we can copy the values from one object to another like copy constructor in C++.
- There are many ways to copy the values of one object into another in Java.

They are:

- By constructor — copy the values of one object into another using Java constructor Student6.java
- By assigning the values of one object into another
- By clone() method of Object class

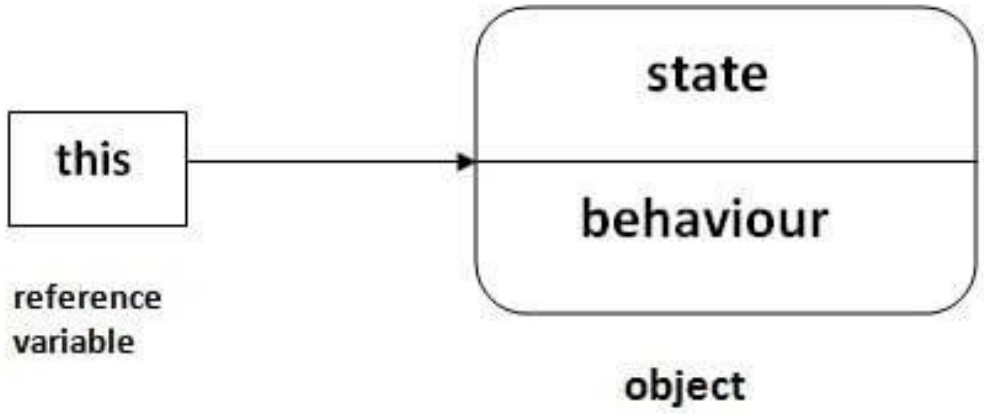
Copying values without constructor

copy the values of one object into another by assigning the objects values to another object. In this case, there is no need to create the constructor. Student7.java

# This keyword

# This keyword

- In Java, **this** is a **reference variable** that refers to the current object.




TestThis1.java TestThis2.java TestThis3.java

# This keyword

## Usage of Java this Keyword

There can be a lot of usage of java this keyword. In java, this is a reference variable that refers to the current object.

01 this can be used to refer current class instance variable.

02 this can be used to invoke current class method (implicity)

03 this() can be used to invoke current class Constructor.

04 this can be passed as an argument in the method call.

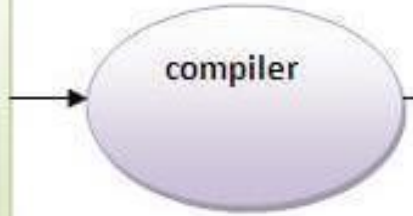
05 this can be passed as argument in the constructor call.

06 this can be used to return the current class instance from the method

# this: to invoke current class method

- One can invoke the method of the current class by using the this keyword.
- If you don't use the this keyword, compiler automatically adds this keyword while invoking the method.

```
class A{

void m(){}

void n(){
m();
}

public static void main(String args[]){

new A().n();

}}
```




```
class A{

void m(){}

void n(){
this.m();
}

public static void main(String args[]){

new A().n();

}}
```

TestThis4.java

# this() : to invoke current class constructor

- The this() constructor call can be used to invoke the current class constructor.
- It is used to reuse the constructor. In other words, it is used for constructor chaining.

**Calling default constructor from parameterized constructor: TestThis5.java**

**Calling parameterized constructor from default constructor: TestThis6.java**

Real usage of this() constructor call TestThis7.java

# this: to pass as an argument in the method

The this keyword can also be passed as an argument in the method.
It is mainly used in the event handling.

```
class S2{
  void m(S2 obj){
  System.out.println("method is invoked");
  }
  void p(){
  m(this);
  }
  public static void main(String args[]){
  S2 s1 = new S2();
  s1.p();
  }
}
```

Application of this that can be passed as an argument:
In event handling (or) in a situation where we have to provide reference of a class to another one. It is used to reuse one object in many methods.

# this: to pass as argument in the constructor call

We can pass the this keyword in the constructor also. It is useful if we have to use one object in multiple classes.

```
class B{
  A4 obj;
  B(A4 obj){
    this.obj=obj;
  }
  void display(){
    System.out.println(obj.data);//using data member of A4 class
  }
}
  class A4{
  int data=10;
  A4(){
   B b=new B(this);
   b.display();
  }
  public static void main(String args[]){
   A4 a=new A4();
  }
}
```

# this keyword can be used to return current class instance

We can return this keyword as an statement from the method. In such case, return type of the method must be the class type (non-primitive). Let's see the example:

Syntax of this that can be returned as a statement

```
return_type method_name(){
return this;
}
```

```
class A{
A getA(){
return this;
}
void msg(){System.out.println("Hello java");}
}
class Test1{
public static void main(String args[]){
new A().getA().msg();
}
}
```

# Garbage Collection

# Java Garbage Collection

- In java, garbage means unreferenced objects.
- Garbage Collection is process of reclaiming the runtime unused memory automatically.
- In other words, it is a way to destroy the unused objects.
- To do so, we were using free() function in C language and delete() in C++.
- But, in java it is performed automatically.
- So, java provides better memory management.

# Advantage of Garbage Collection

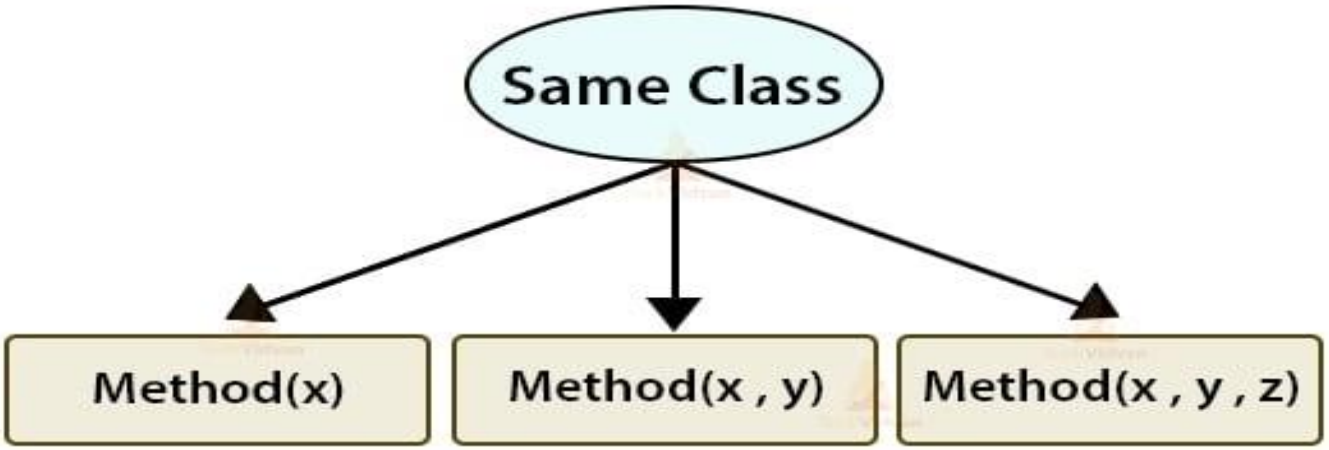
- It makes java memory efficient because garbage collector removes the unreferenced objects from heap memory.
- It is automatically done by the garbage collector (a part of JVM) so we don't need to make extra efforts.

# Methods and Classes

# Overloading methods

- Method Overloading allows different methods to have the same name, but different signatures where the signature can differ by the number of input parameters or type of input parameters, or a mixture of both.
- Method overloading in Java is also known as Compile-time Polymorphism, Static Polymorphism, or Early binding.
- In Method overloading compared to the parent argument, the child argument will get the highest priority.

# Method Overloading in Java




There are two ways to overload the method in java

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

# changing no. of arguments

```
class Adder{
static int add(int a,int b)
{
return a+b;
}
static int add(int a,int b,int c)
{
return a+b+c;}
}
class TestOverloading1
{
public static void main(String[] args)
{
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(11,11,11));
}
}
```

# changing data type of arguments

```
class Adder
{
static int add(int a, int b)
{
return a+b;
}
static double add(double a, double b)
{
return a+b;
}
}
class TestOverloading2
{
public static void main(String[] args)
{
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(12.3,12.6));
}
}
```

Output

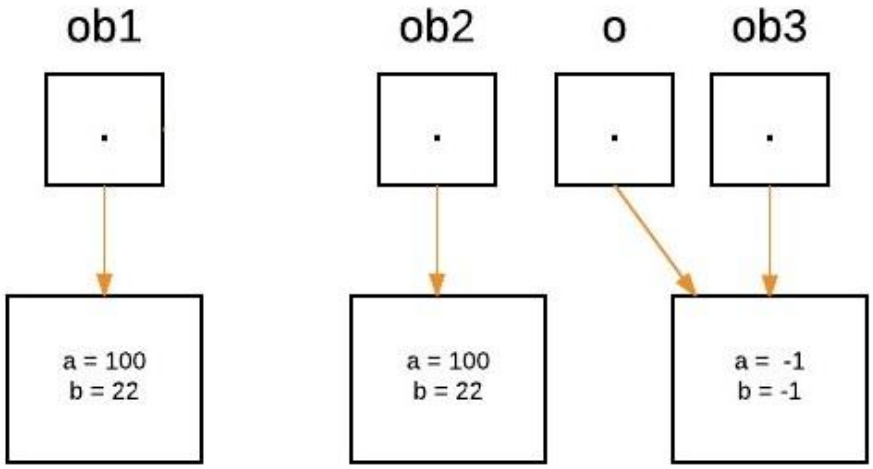
22

24.9

# Objects as Parameters

- 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.
- Any modifications made to the object within the method will have an impact on the original object.

The general form to demonstrate "object as parameter" in Java is as follows:



```
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
}
```

# Argument Passing

- Primitive variables are directly stored in stack memory.
- Whenever any variable of primitive data type is passed as an argument, the actual parameters are copied to formal arguments and these formal arguments accumulate their own space in stack memory.
- Each parameter consists of two parts: type name and variable name. A type name followed by a variable name defines the type of value that can be passed to a method when it is called. It is also often called formal parameter.

```
public class Sum {
public static void main(String[] args)
{
// Creating an object of class Sum.
   Sum obj = new Sum();

// Calling the sum() method by passing argument values to
the method's parameters.
// The returned value is stored in a variable named x of
type int.
   int x = obj.sum(20, 10);
   System.out.println(x);
}
// Method declaration with a return statement.
   int sum (int a, int b)
   {
      int s = a + b;
      return s;
   }
}
```

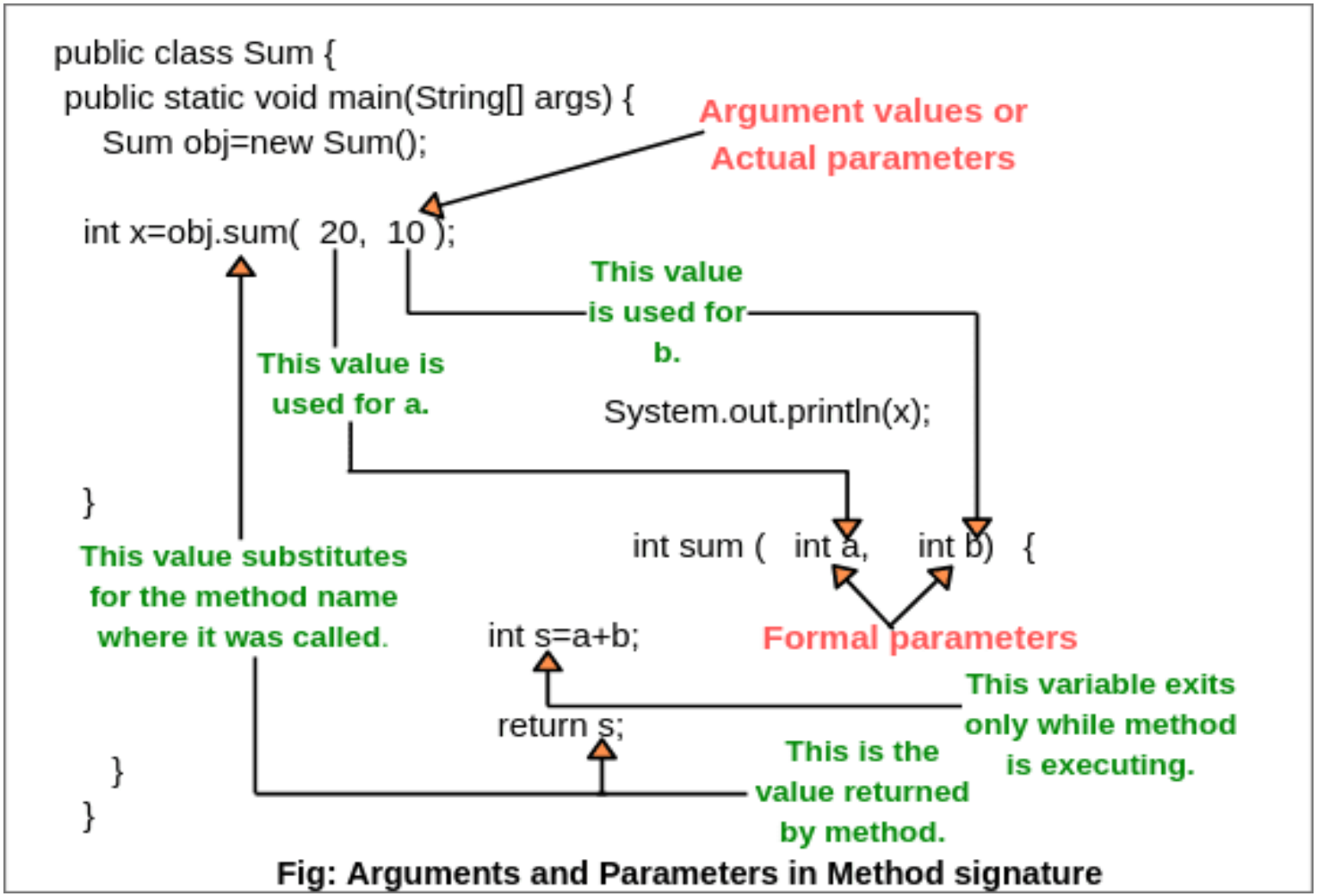



**Fig: Arguments and Parameters in Method signature**

# Returning Objects

- A method can have the class name as its return type.
- It must return the object of the exact class or its subclass.
- An interface name can also be used as a return type but the returned object must implement methods of that interface.
- The return statement is used for returning a value when the execution of the block is completed. The return statement inside a loop will cause the loop to break and further statements will be ignored by the compiler.

```
public class SampleReturn1
{ .
    /* Method with an integer return type and no arguments */
    public int CompareNum()
    {
        int x = 3;
        int y = 8;
        System.out.println("x = " + x + "\ny = " + y);
        if(x>y)
            return x;
        else
            return y;
    }

    /* Driver Code */
    public static void main(String ar[])
    {
        SampleReturn1 obj = new SampleReturn1();
        int result = obj.CompareNum();
        System.out.println("The greater number among x and y is: " + r
esult);
    }
}
```

**Output:**

```
x = 3
y = 8
The greater number among x and y is: 8
```

# Recursion

Recursion in java is a process in which a method calls itself continuously. A method in java that calls itself is called recursive method.

**Syntax:**

```
Return type method name() {
//code to be executed
Method name();//calling same method
}
```

```
public class RecursionExample2 {
static int count=0;
static void p(){
count++;
if(count<=5){
System.out.println("hello "+count);
p();
}
}
public static void main(String[] args) {
p();
}
}
```

Output
hello 1
hello 2
hello 3
hello 4
hello 5

# Access Control

- There are two types of modifiers in Java: access modifiers and non-access modifiers.

- The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class.
- By applying the access modifier, one can change the access level of fields, constructors, methods, and class.

# Access Control

There are four types of Java access modifiers:

1. Private: The access level of a private modifier is only **within the class**. It cannot be accessed from outside the class. Simple.java
2. Default: The access level of a default modifier is only **within the package**. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default. A.java & B.java (javac -d . B.java, java mypack.B)
3. Protected: The access level of a protected modifier is **within the package and outside the package through child class**. If you do not make the child class, it cannot be accessed from outside the package. C.Java & D.java (javac –d . C.java , javac –d . D.java, java pack1.D)
4. Public: The access level of a public modifier is **everywhere**. It can be accessed from within the class, outside the class, within the package and outside the package. pubA.java & pubB.java

# Access modifiers in Java

- There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc.

| Access Modifier | within class | within package | outside package by subclass only | outside package |
|---|---|---|---|---|
| **Private** | Y | N | N | N |
| **Default** | Y | Y | N | N |
| **Protected** | Y | Y | Y | N |
| **Public** | Y | Y | Y | Y |

# Understanding Static

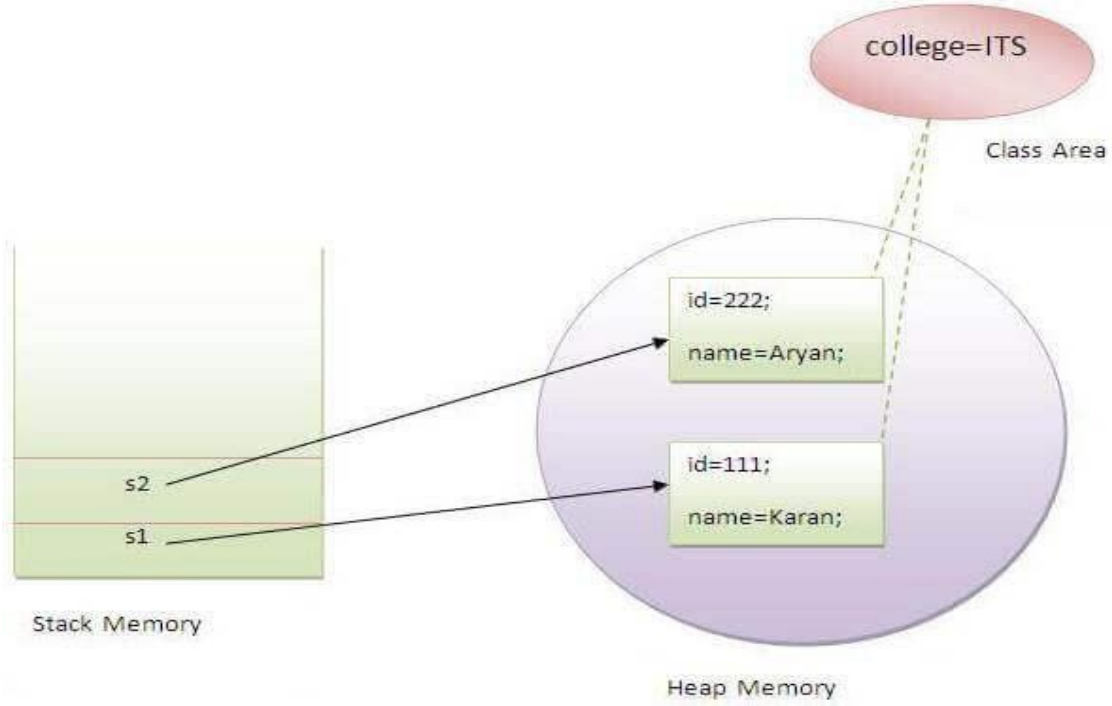
The **static keyword** in Java is used for memory management mainly. We can apply static keyword with variables, methods, blocks and nested classes. The static keyword belongs to the class than an instance of the class.

The static can be:

- Variable (also known as a class variable)
- Method (also known as a class method)
- Block
- Nested class

# Understanding Static

- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.



```
class Counter2{
static int count=0;//will get memory only once and retain its va
lue

Counter2()
{
count++;//incrementing the value of static variable
System.out.println(count);
}

public static void main(String args[])

{
//creating objects
Counter2 c1=new Counter2();
Counter2 c2=new Counter2();
Counter2 c3=new Counter2();
}
}
```

Output

1

2

3

# Introducing Final

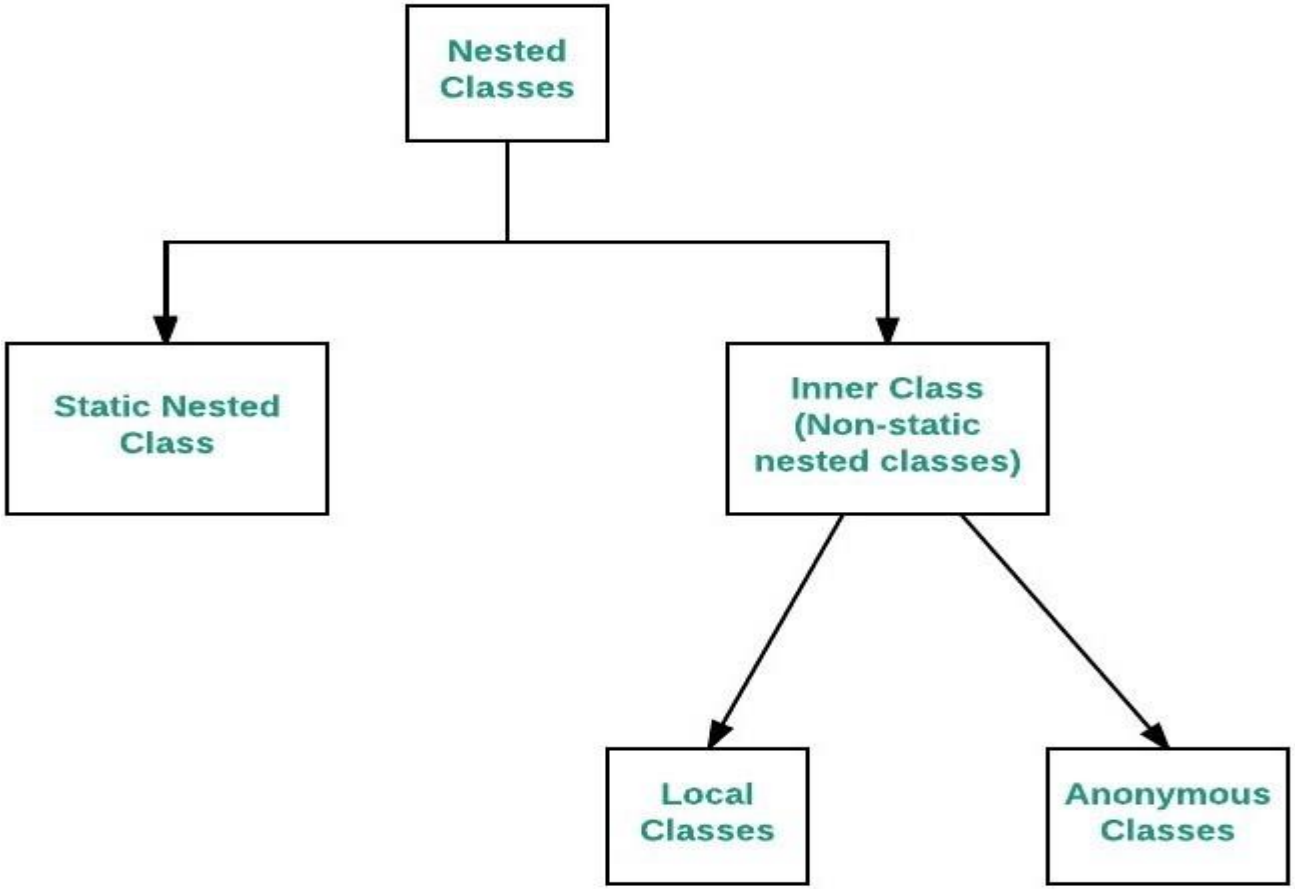
The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

- Variable
- Method
- class

- The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable.
- It can be initialized in the constructor only.
- The blank final variable can be static also which will be initialized in the static block only..

# Introducing Nested and Inner Classes

- Defining a class within another class, such classes are known as *nested* classes.
- They enable you to logically group classes that are only used in one place, thus this increases the use of encapsulation and creates more readable and maintainable code.
- Nested classes are divided into two categories:
- **static nested class:** Nested classes that are declared *static* are called static nested classes.
- **inner class:** An inner class is a non-static nested class


Nested Classes
Static Nested Class
Inner Class (Non-static nested classes)
Local Classes
Anonymous Classes


**Inheritance**: inheritance basics, using super, creating multi level hierarchy, method overriding.

# 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.

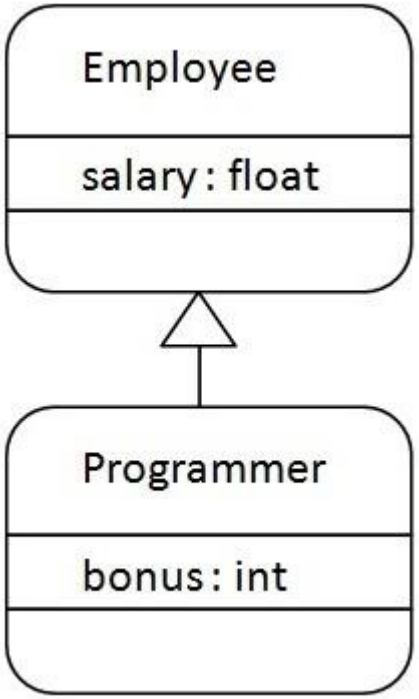
# The syntax of Java Inheritance

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

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

- In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

# Java Inheritance Example



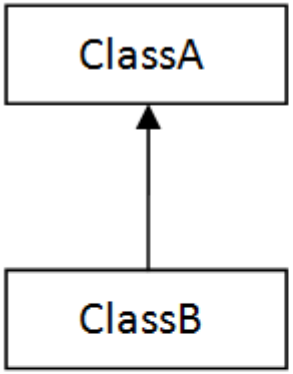

- Programmer is the subclass and Employee is the superclass.
- The relationship between the two classes is **Programmer IS-A Employee**.
- It means that Programmer is a type of Employee.
- Programmer.java

Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

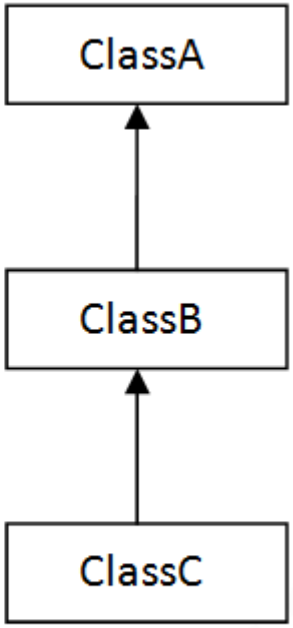
# Types of inheritance in java

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

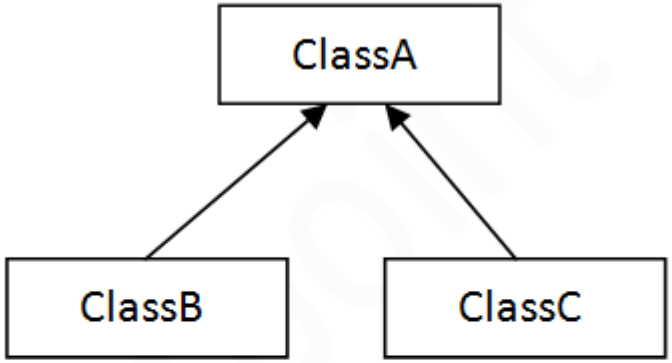
In java programming, multiple and hybrid inheritance is supported through interface only.




1) Single



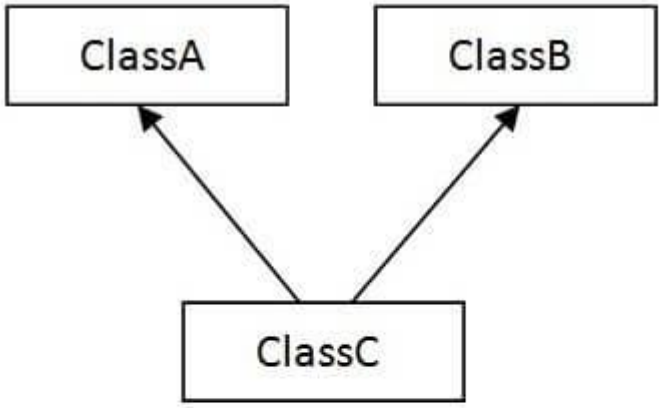

2) Multilevel



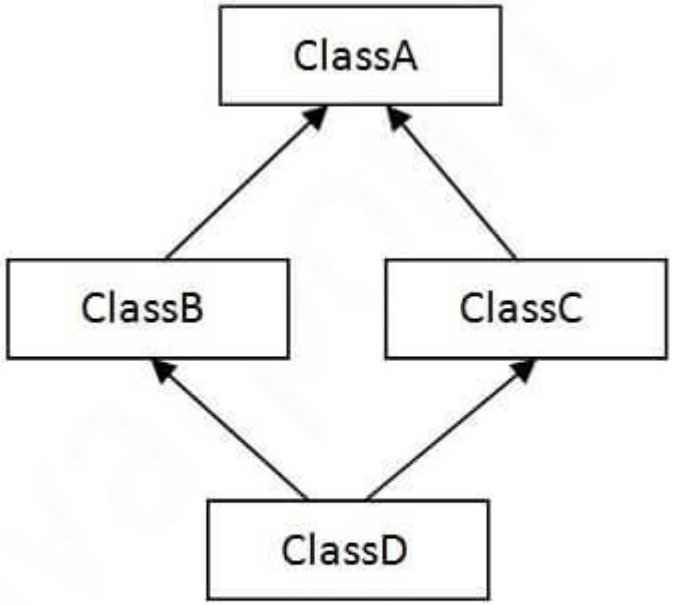

3) Hierarchical

# Types of inheritance in java

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




4) Multiple




5) Hybrid

# 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.

```
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();
}}
```

# 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.

*TestInheritance2.java*

# 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.

*TestInheritance3.java*

# Why multiple inheritance is not supported in java?

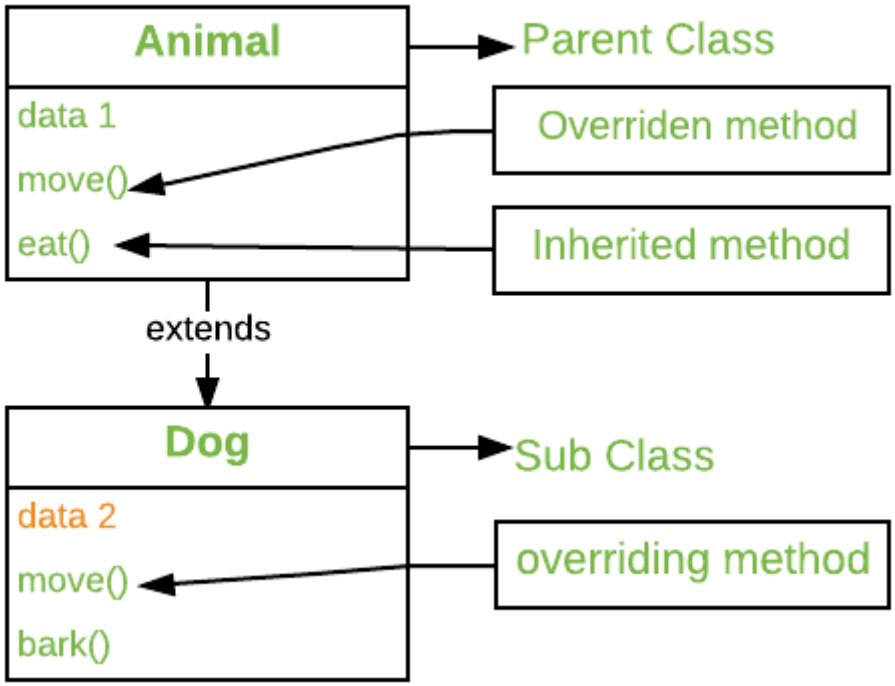
To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

- Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.
- Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

```
class A{
void msg(){System.out.println("Hello");}
}
class B{
void  msg(){System.out.println("Welcome");
}
}
class C extends A,B{//suppose if it were

 public static void main(String args[]){
   C obj=new C();
    obj.msg();//Now which msg() method wo
uld be invoked?
}
}
```

# 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**.



# Method Overriding in Java

- 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.**

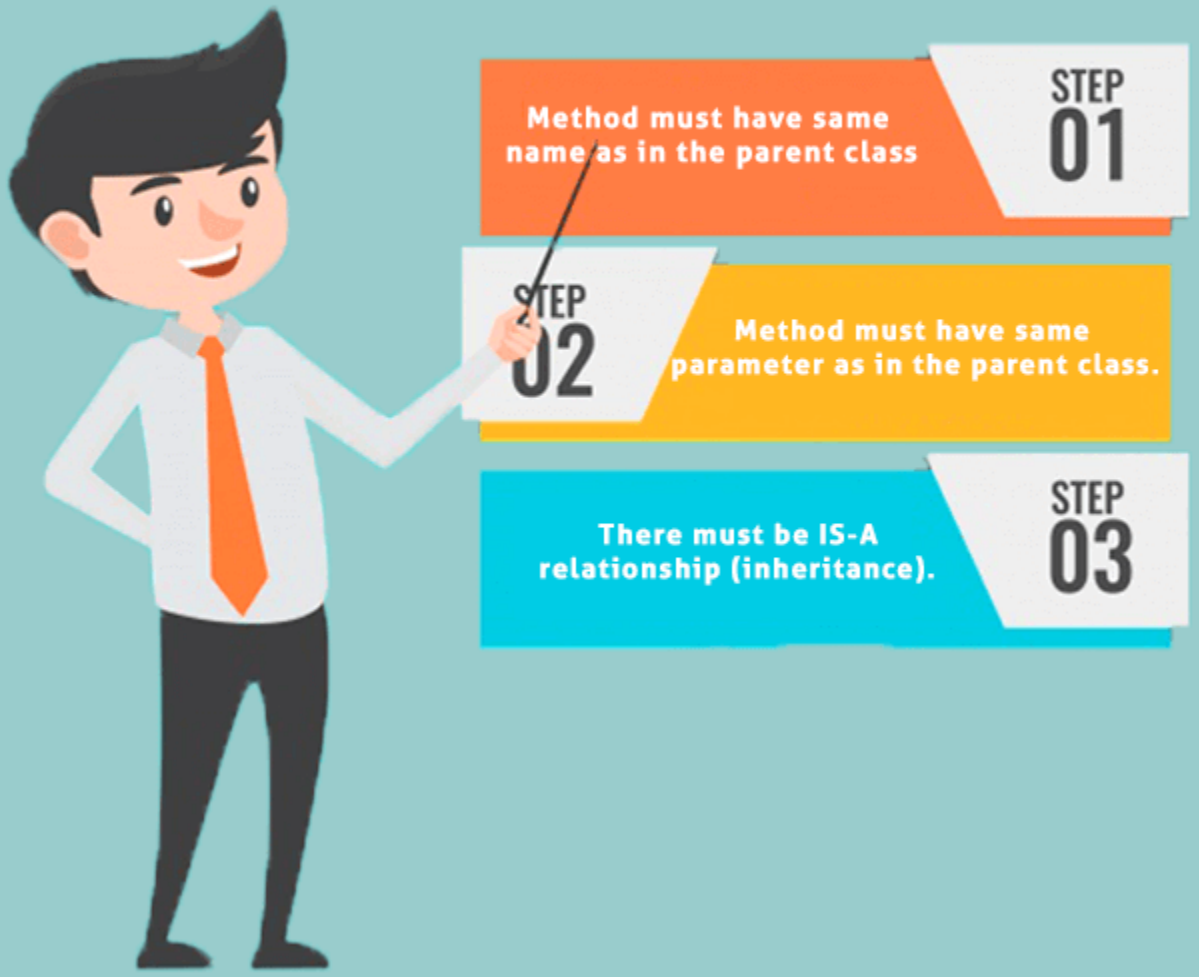
## Usage of Java Method Overriding

- 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).

# Rules for Java Method Overriding

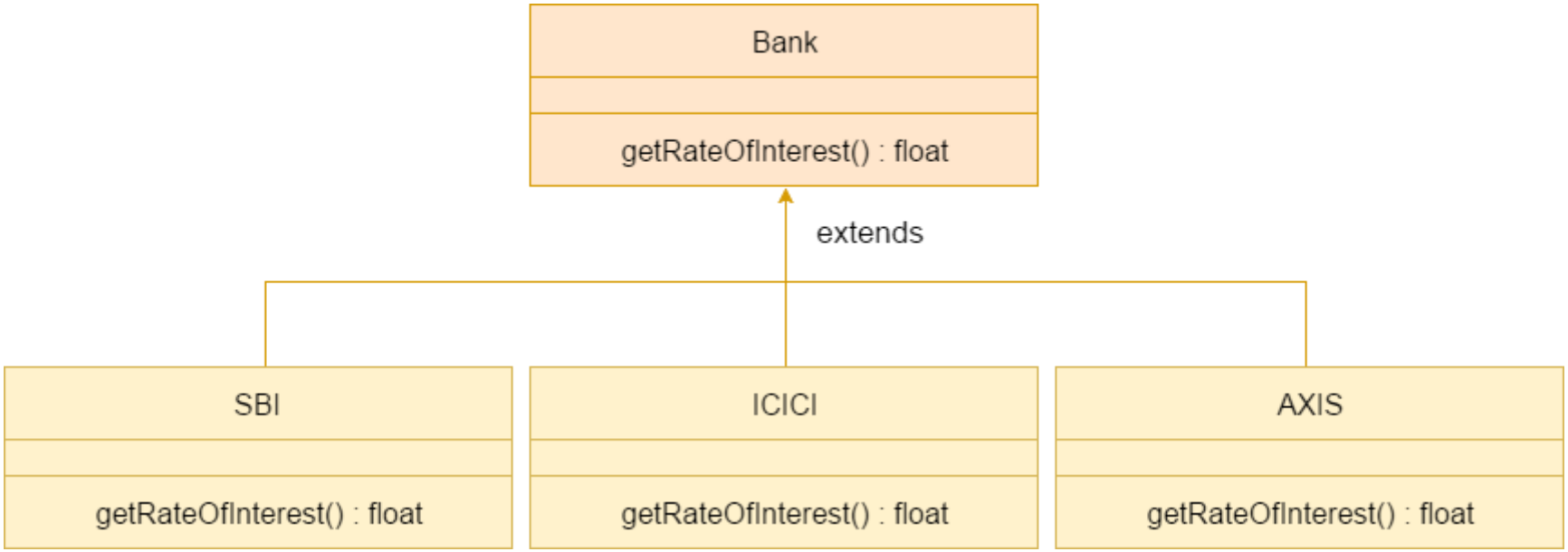


# Examples

Example program without method overriding Bike.java

- 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**.
- Bike2.java

# A real example of Java Method Overriding




Test2.java

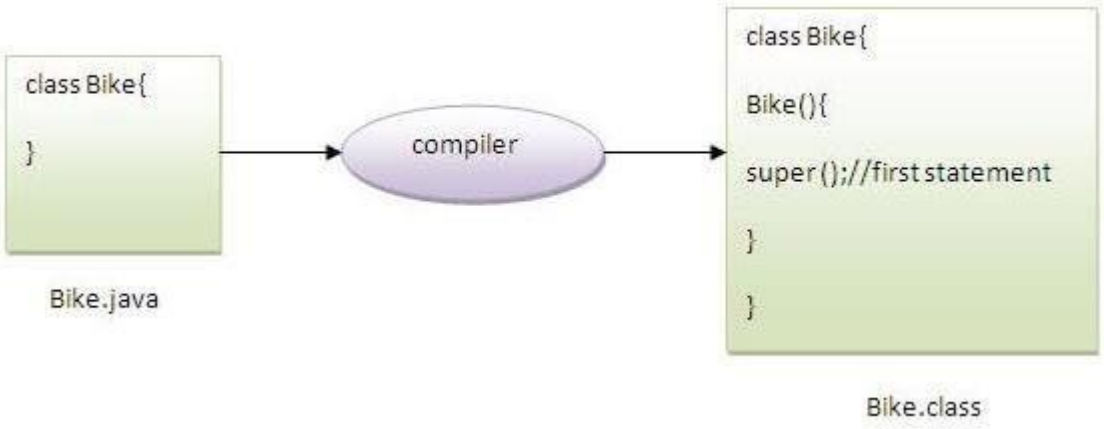
# What is Overloading and Overriding?

- When two or more methods in the same class have the same name but different parameters, it's called **Overloading**.
- Method Overloading is a **Compile time polymorphism.**
- When the method signature (name and parameters) are the same in the superclass and the child class, it's called **Overriding**.
- Method overriding is one of the way by which java achieve **Run Time Polymorphism**.

# Super keyword in JAVA

- The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.
- Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

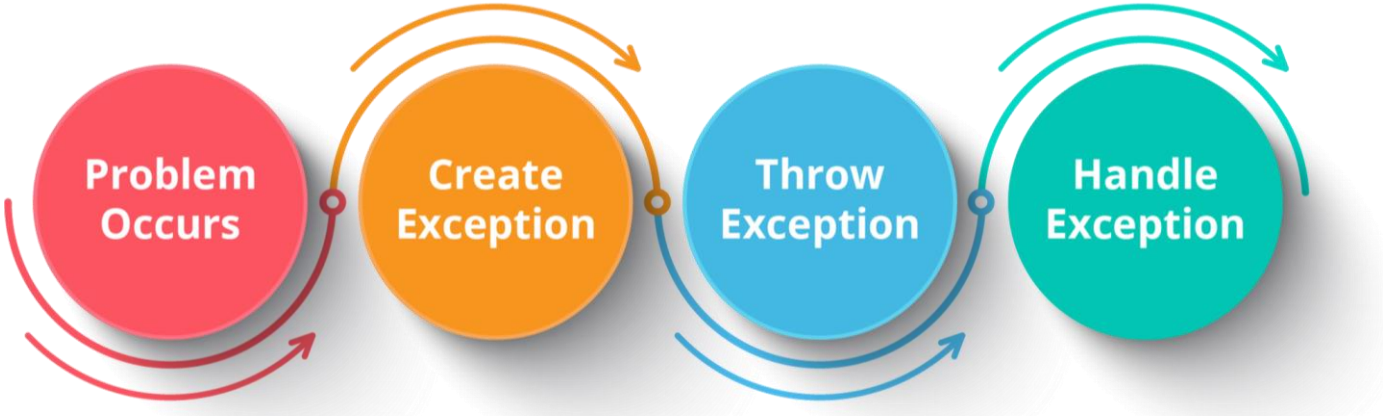
1. Super is used to refer immediate parent class instance variable TestSuper1.java
2. Super can be used to invoke parent class method TestSuper2.java
3. Super is used to invoke parent class constructor TestSuper3.java




Note: super() is added in each class constructor automatically by compiler if there is no super() or this().

**Another example of super keyword where super() is provided by the compiler implicitly. TestSuper4.java**

super example: real use TestSuper5.java




**Exception handling**: Exception handling in Java.

# Exception Handling in Java

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.

**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.

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

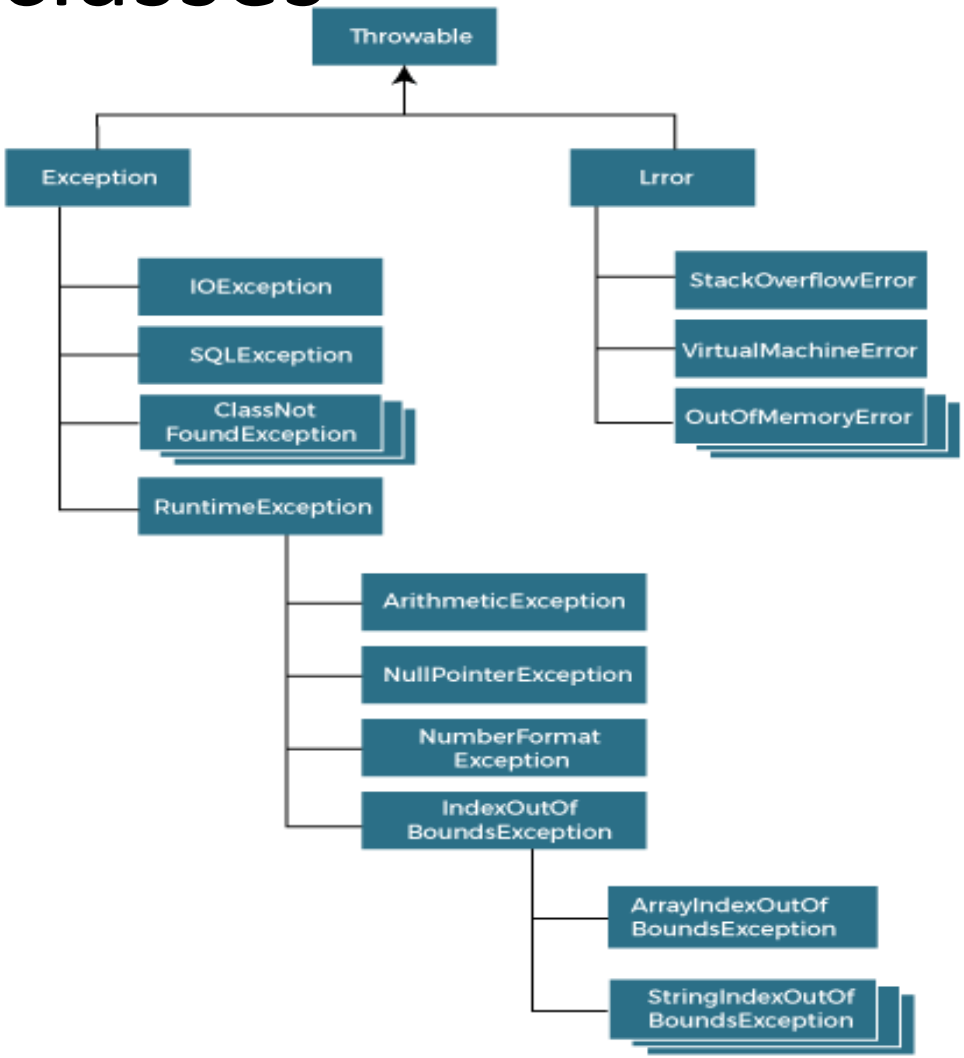
# Advantage of Exception Handling

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.

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

# 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**.



# 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

# Java Exception Keywords

| 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. |

# 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 Handling Example

```
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...");
  }
}
```

# 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.

```
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.

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

# Common Scenarios of Java Exceptions

## 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.

```
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.

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

# 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){}
```

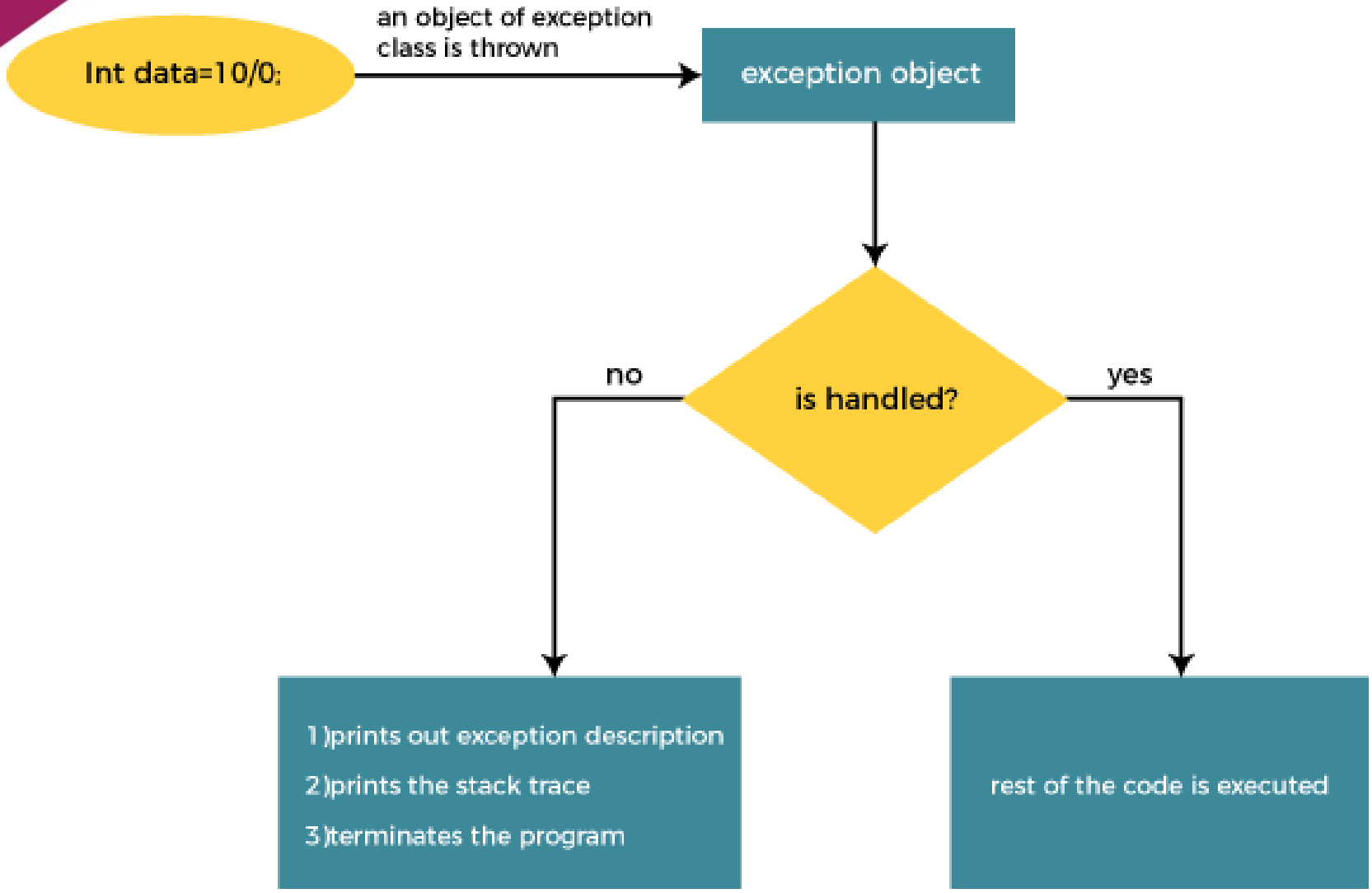
### Syntax of try-finally block

```
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.


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

```
public class TryCatchExample1 {

    public static void main(String[] args) {

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

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

    }

}
```

The **rest of the code** is not executed (in such case, the **rest of the code** statement is not printed).

# Solution by exception handling

```
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");
   }

}
```

The **rest of the code** is executed, i.e., the **rest of the code** statement is printed.

The code in a try block that will not throw an exception

**TryCatchExample3.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);
      }

   }

}
```

If an exception occurs in the try block, the rest of the block code will not execute.

Handle the exception using the parent class exception.

**TryCatchExample4.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");
   }

}
```

An example to print a custom message on exception

**TryCatchExample5.java**

```
public class TryCatchExample5 {

    public static void main(String[] args) {
        try
        {
        int data=50/0; //may throw exception
        }
            // handling the exception
        catch(Exception e)
        {
                // displaying the custom message
            System.out.println("Can't divided by zero");
        }
    }

}
```

An example to resolve the exception in a catch block

TryCatchExample6.java

Example to handle the generated exception (Arithmetic Exception) with a different type of exception class (ArrayIndexOutOfBoundsException). TryCatchExample8.java

An example to handle another unchecked exception.

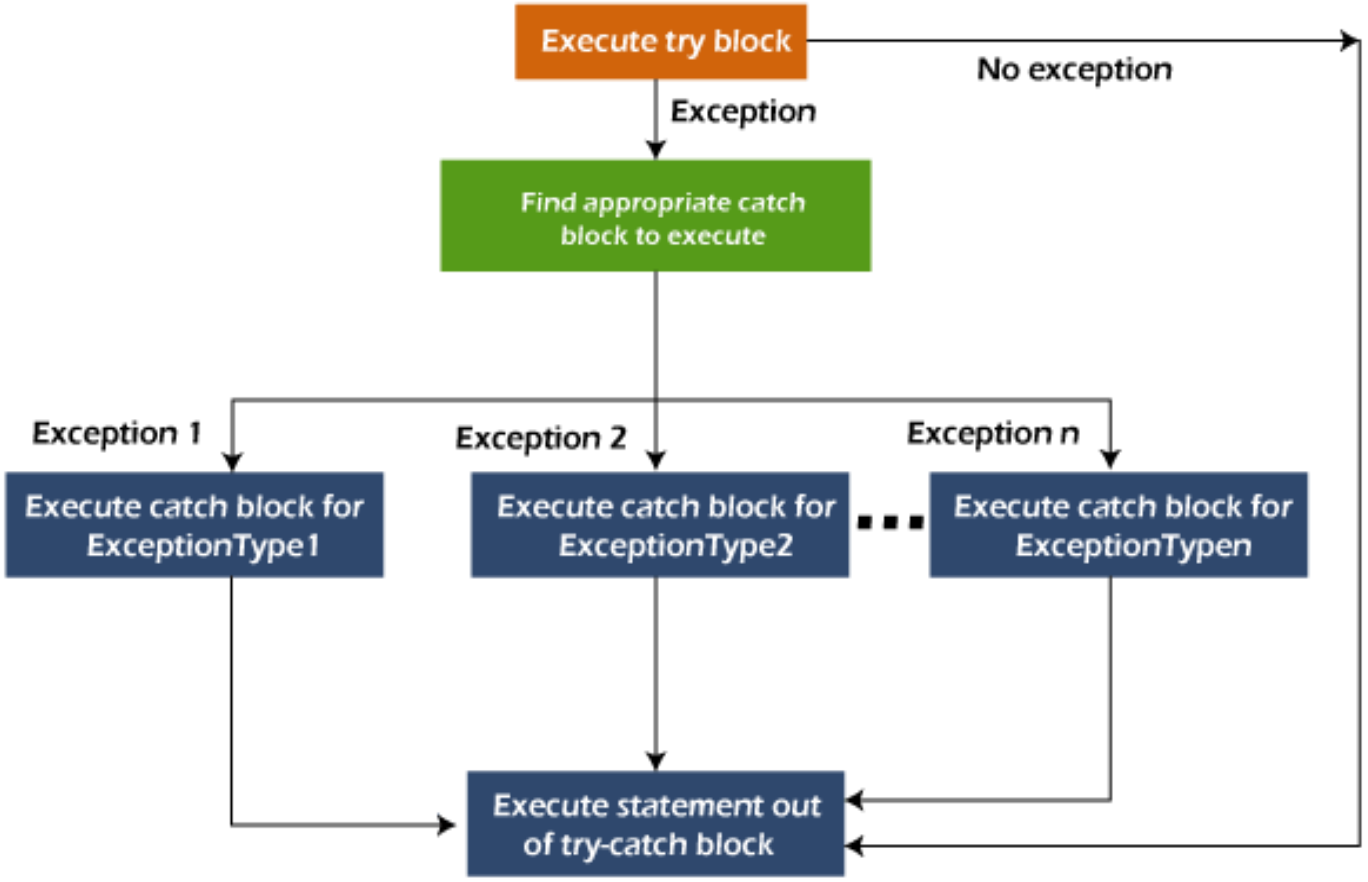
TryCatchExample9.java

An example to handle checked exception. TryCatchExample10.java

# Java Catch Multiple Exceptions

**Java Multi-catch block**

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.



## A simple example of java multi-catch block. MultipleCatchBlock1.java

In this example, try block contains two exceptions. But at a time only one exception occurs and its corresponding catch block is executed. MultipleCatchBlock2.java MultipleCatchBlock3.java

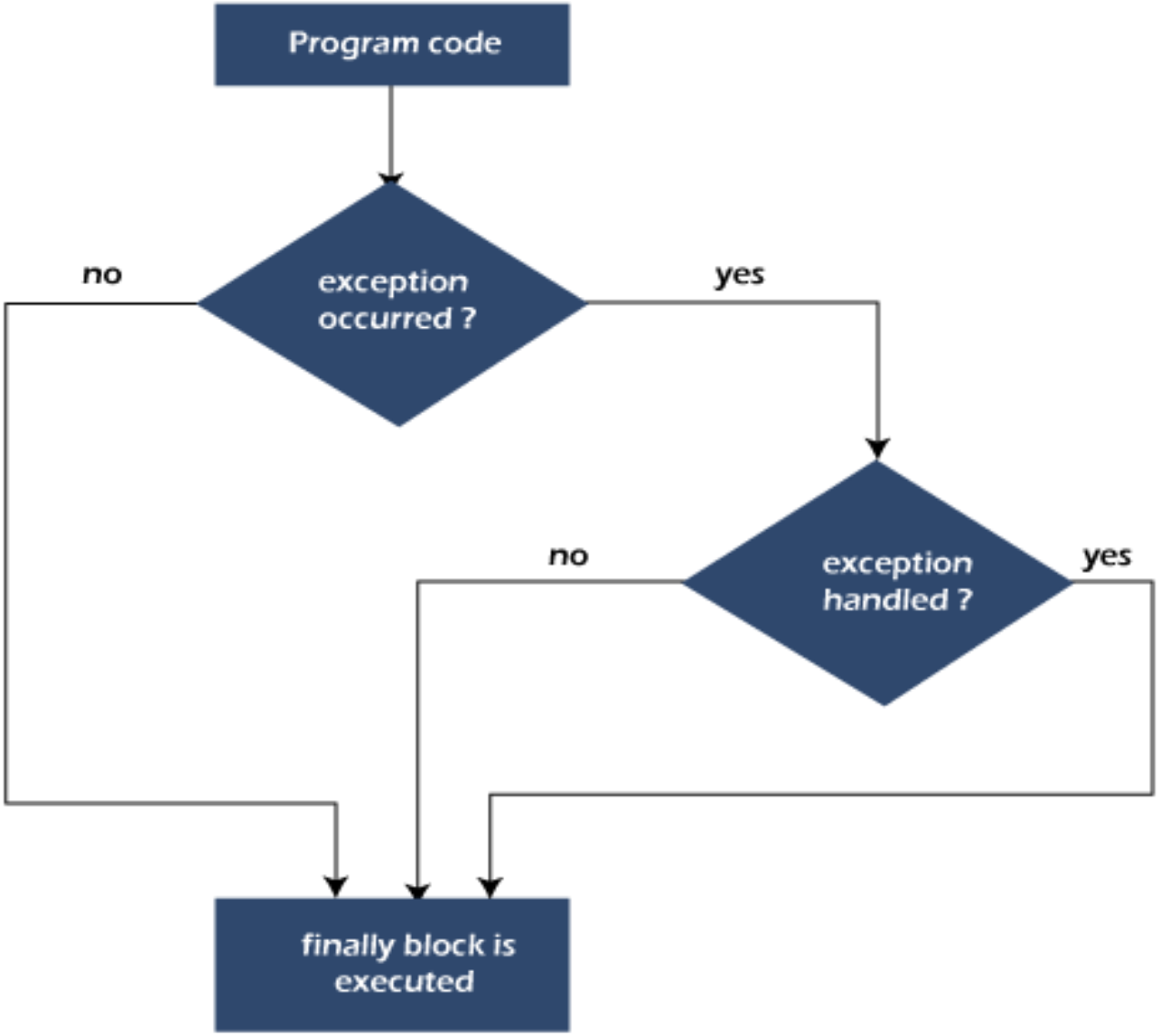
In this example, we generate NullPointerException, but didn't provide the corresponding exception type. In such case, the catch block containing the parent exception class **Exception** will invoked. MultipleCatchBlock4.java

Example to handle the exception without maintaining the order of exceptions (i.e. from most specific to most general). MultipleCatchBlock5.java

# Java finally block

- **Java finally block** is a block used to execute important code such as closing the connection, etc.
- Java finally block is always executed whether an exception is handled or not.
- Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.
- The finally block follows the try-catch block.

# Flowchart of finally block



## Why use Java finally block?

- finally block in Java can be used to put "**cleanup**" code such as closing a file, closing connection, etc.
- The important statements to be printed can be placed in the finally block.

Usage of Java finally

Case 1: When an exception does not occur. TestFinallyBlock.java

Case 2: When an exception occur but not handled by the catch block TestFinallyBlock1.java

Case 3: When an exception occurs and is handled by the catch block. TestFinallyBlock2.java

# 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.
- 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.
- Can throw either checked or unchecked exceptions in Java by throw keyword.
- It is mainly used to throw a custom exception.
- 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

## Example 2: Throwing Checked Exception TestThrow2.java

## Example 3: Throwing User-defined Exception

- Exception is everything else under the Throwable class.

TestThrow3.java