www.kiet.edu
Delhi-NCR, Ghaziabad
Established: 1998

KIET
GROUP OF INSTITUTIONS
(An Autonomous Institute)
Connecting Life with Learning

A+
NAAC
GRADE

Object Oriented Programming with Java (Subject Code: CS301L)

Unit 1

Unit 1

- ❑Introduction to Java, JVM, JRE
- ❑Java Environment
- ❑Java Source File Structure and Compilation.
- ❑Defining Classes in Java
- ❑Constructors, Methods, Access Specifies
- ❑Static Members, Final Members,
- ❑Class, Object, Abstraction, Inheritance, Encapsulation, Polymorphism
- ❑Interface,
- ❑Exceptions: Use of try, catch, finally, throw, throws, In-built and User Defined Exceptions, Checked and Un-Checked Exceptions.

Why Java

- ### Java is Easy to Learn

Java is beginner-friendly and one of the most popular programming languages among new developers.

- ### Java is Versatile

Java follows the ‘write once and run anywhere’ principle and can be used for programming applications using different platforms.

- ### Java is Open Source

Most of Java’s features are open-source; this makes building applications cheap and easy.

❑ Java is Object-Oriented

Java is an object-oriented programming language and this makes it scalable and flexible.

❑ Java is Scalable

Java is used everywhere, including desktops, mobile, applications, and so on. It can effectively run on any operating system and is ideal for building applications.

❑ Java is Platform-Independent

Java has the ability to easily move across platforms and can be run similarly on different systems.

❑ Java Has a Rich API

Java has a rich Application Programming Interface (API) system that includes packages, interfaces, and classes, along with their methods and fields.

❑ Java is Free of Cost

Java is a free-to-download software on Oracle Binary Code License (BCL), enabling beginners to develop applications easily and learn Java programming effectively.

History of Java

- Java is an Object-Oriented programming language developed by **James Gosling** in the early 1990s.
- The team initiated this project to develop a language for digital devices such as set-top boxes, television, etc.
- Originally C++ was considered to be used in the project but the idea was rejected for several reasons.
- James Gosling and his team called their project "**Greentalk**" and its file extension was **.gt** and later became to known as "**OAK**".

Why “Oak”?

- The name **Oak** was used by **Gosling** after an **oak tree** that remained outside his office.
- But they had to later rename it as “**JAVA**” as it was already a trademark by **Oak Technologies**.
- **Java** name was decided after much discussion since it was so unique.
- Gosling came up with this name while having a coffee near his office.

- Java was created on the principles like **Robust, Portable, Platform Independent, High Performance, Multithread, etc.** and was called one of the **Ten Best Products of 1995** by the **TIME MAGAZINE**.

History of various Java versions

VERSION	RELEASE DATE
JDK Beta	1995
JDK 1.0	January 1996
JDK 1.1	February 1997
J2SE 1.2	December 1998
J2SE 1.3	May 2000
J2SE 1.4	February 2002
J2SE 5.0	September 2004
JAVA SE 6	December 2006
JAVA SE 7	July 2011
JAVA SE 8	March 2014
JAVA SE 9	September 2017
JAVA SE 10	March 2018
JAVA SE 11	September 2018
JAVA SE 12	March 2019

Latest Versions of Java

- Java SE 21 (LTS) September, 19th 2023
- Java 22 was released on March 19, 2024.
- Java 24 was released on March 18, 2025

OOPs (Object Oriented Programming System)

- **OOPs (Object-Oriented Programming System)** is a programming paradigm based on the concept of **objects**, which can contain **data** (fields, also called attributes or properties) and **methods** (functions) that operate on that data.
- Its main goal is to structure programs so that they are easier to develop, maintain, and reuse.

❑Class

A class is a **template** or **blueprint** that describes the behaviors (methods) and states (variables) that objects of its type support.

❑Object

Objects have states and behaviors.

Example:

States: color, name, breed (for a dog)

Behaviors: wagging, barking, eating

Note: An object is an instance of a class.

Key Concepts of OOP

There are four main pillars of OOP:

- **Encapsulation**

  Wrapping data (variables) and methods (functions) into a single unit called an object. Access to data is restricted using access modifiers (private, public, protected).

- **Abstraction**

  Hiding implementation details and showing only the essential features. Achieved using **abstract classes** and **interfaces**.

- **Inheritance**

  Acquiring properties and behaviors of one class into another.

  Promotes **code reusability**.

- **Polymorphism**

  The ability of an object to take many forms (same method name, different behavior).

  Types:

  - **Compile-time (Method Overloading)**
  - **Runtime (Method Overriding)**

Features of Java

- ❑ **Object Oriented** : In java everything is an Object.
- ❑ **Platform independent:** Unlike many other programming languages including C and C++ when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code.
- ❑ **Simple :**Java is designed to be easy to learn. If you understand the basic concept of OOP java would be easy to master.
- ❑ **Secure :** With Java's secure feature it enables to develop virus-free, tamper-free systems.

- ❑**Architectural- neutral** :Java compiler generates an architecture-neutral object file format which makes the compiled code to be executable on many processors.
- ❑**Portable** :being architectural neutral and having no implementation dependent aspects of the specification makes Java portable.
- ❑**Robust** :Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking.
- ❑**Multi-threaded** : With Java's multi-threaded feature it is possible to write programs that can do many tasks simultaneously.

- **Interpreted** :Java byte code is translated on the fly to native machine instructions and is not stored anywhere.
- **Distributed** :Java is designed for the distributed environment of the internet.

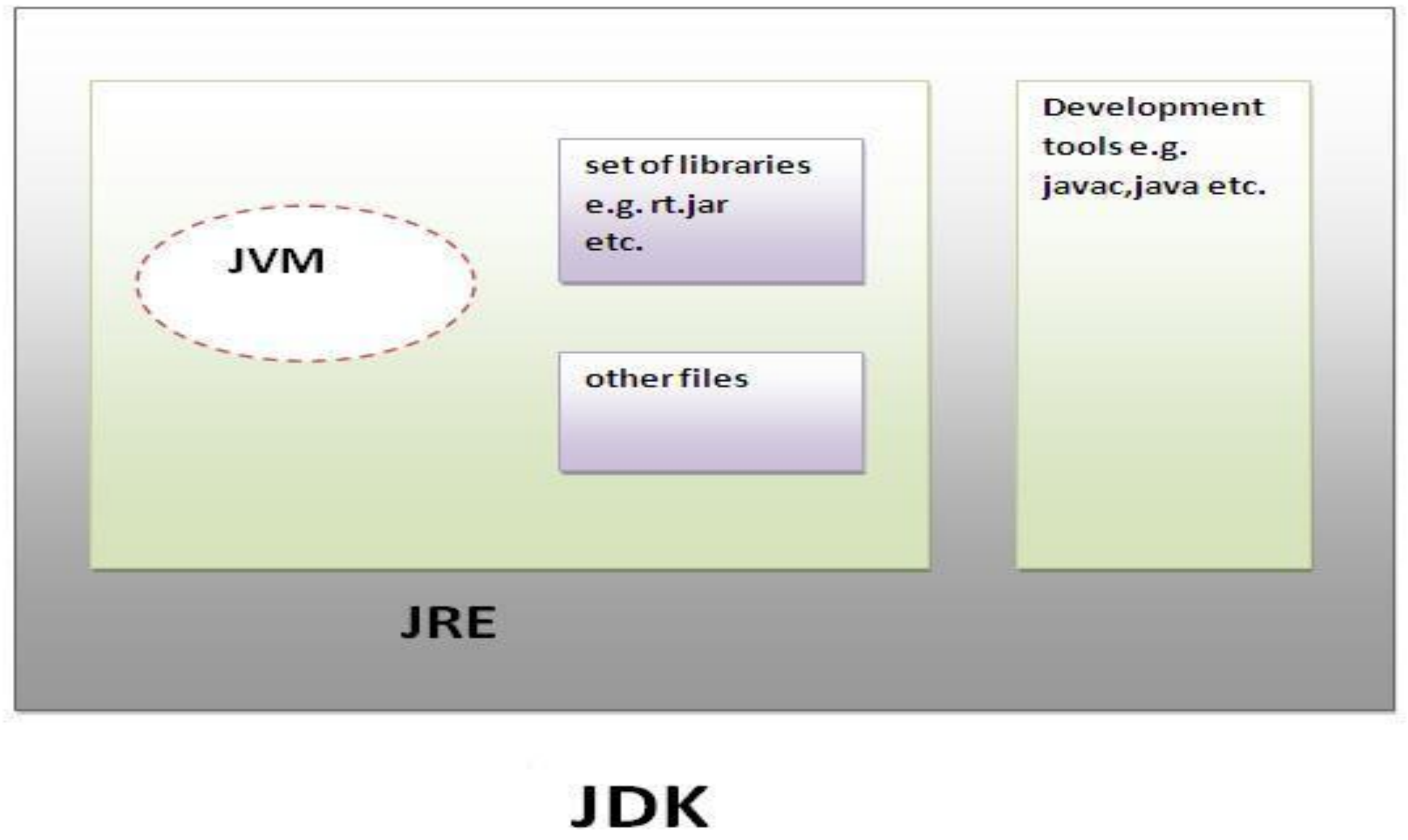
JDK

The Java Development Kit (JDK) is a cross-platformed software development environment that offers a collection of tools and libraries necessary for developing Java-based software applications.

It is a core package used in Java, along with the JVM (Java Virtual Machine) and the JRE (Java Runtime Environment).

JDK

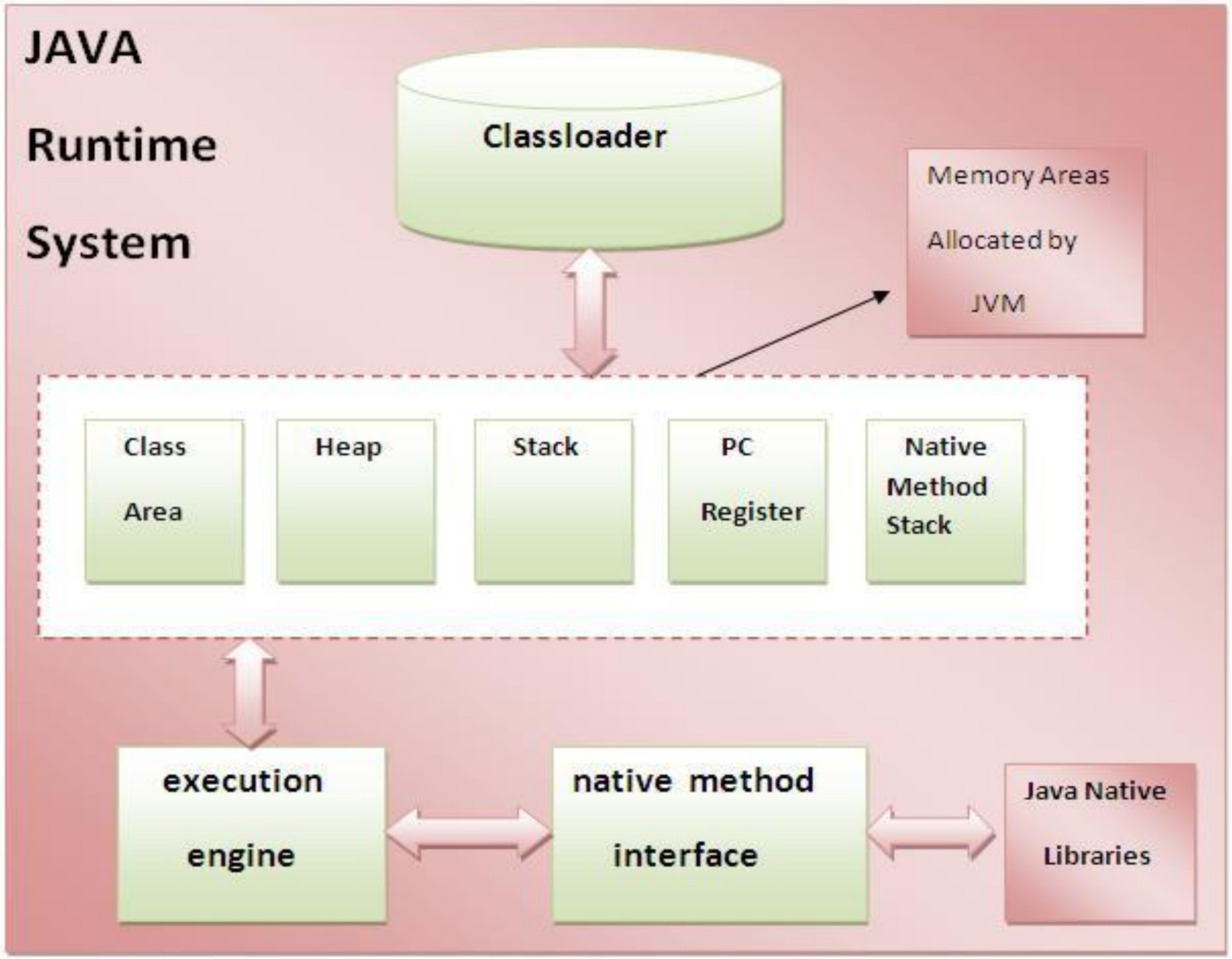
- JDK is an acronym for Java Development Kit.It physically exists.It contains JRE + development tools.


JVM
set of libraries e.g. rt.jar etc.
other files
Development tools e.g. javac,java etc.
JRE
JDK

JVM

- JVM (Java Virtual Machine) is an abstract machine.
- It is a specification that provides runtime environment in which java byte code can be executed.
- JVMs are available for many hardware and software platforms.
- JVM, JRE and JDK are platform dependent because configuration of each OS differs. But, Java is platform independent.

Internal Architecture of JVM



- ❑ **Classloader**

Classloader is a subsystem of JVM that is used to load class files.

- ❑ **Class(Method) Area**

Class(Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.

- ❑**Heap**

It is the runtime data area in which objects are allocated.

- ❑**Stack**

It holds local variables and partial results, and plays a part in method invocation and return.

- ❑ **Program Counter Register**

It contains the address of the Java virtual machine instruction currently being executed.

- ❑ **Native Method Stack**

It contains all the native methods used in the application.

- ❑ **Execution Engine**

It contains:

i) **A virtual processor**

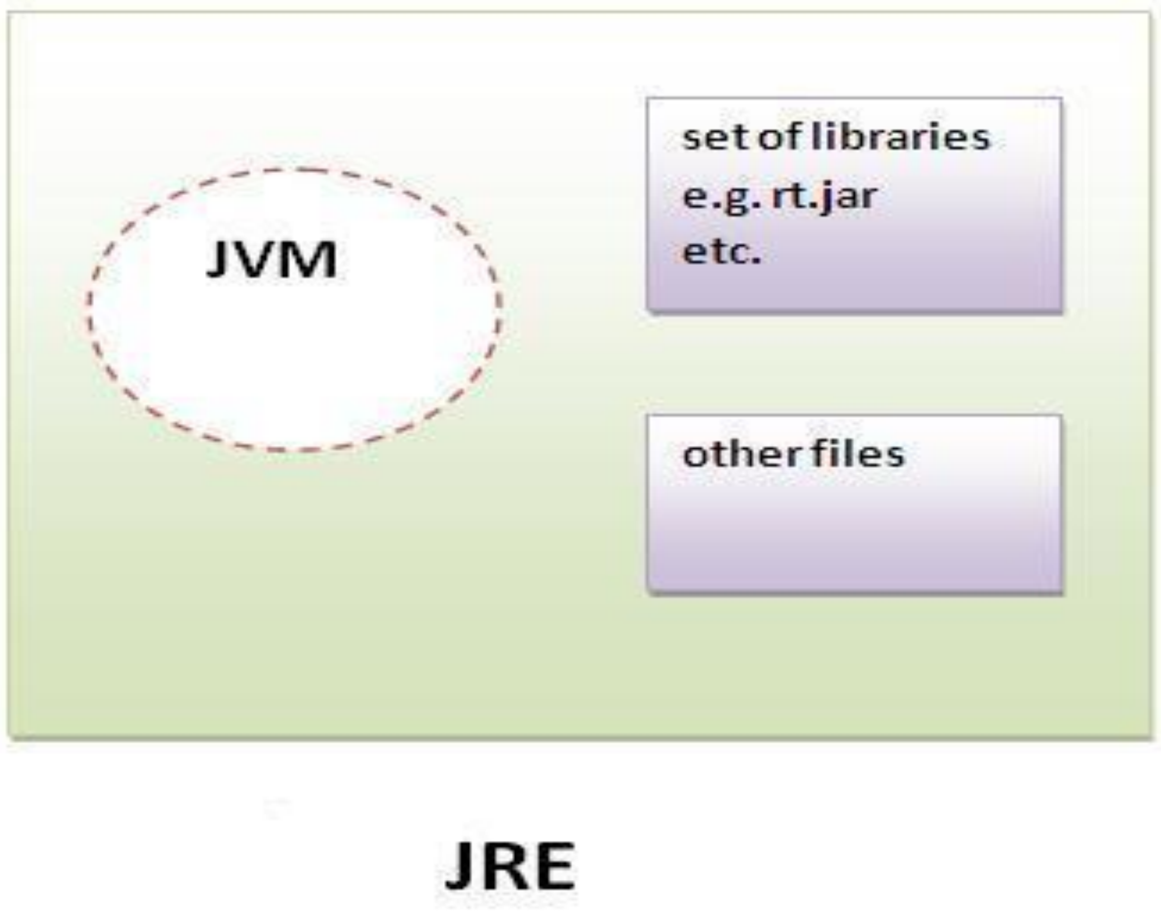
ii) **Interpreter:** Read bytecode stream then execute the instructions.

iii) **Just-In-Time(JIT) compiler:** It is used to improve the performance.JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed.

JRE

- JRE is an acronym for Java Runtime Environment.
- It is used to provide runtime environment. It is the implementation of JVM.
- It physically exists.
- It contains set of libraries + other files that JVM uses at runtime.

JRE



Java Source File Structure

- In Java, a source file typically follows a specific structure to define classes, interfaces, and other elements of the program.
- Package Declaration (Optional):

  The package declaration is used to organize related classes and interfaces into a package. It is the first non-comment line in the file and is optional.

  For example: package com.example.myapp;

- **Import Statements (Optional):** Import statements are used to bring in classes or entire packages from other packages to use in the current source file. They appear after the package declaration (if present) and before the class declaration.

For example:

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

- **Class Declaration:** A Java source file can contain one public class (with the same name as the file) and any number of non-public classes. The class declaration consists of the class keyword followed by the class name and optional modifiers (e.g., public, abstract, final). For example:

- **Interface Declaration (Optional):** Similar to classes, a Java source file can also contain interfaces. The interface declaration consists of the interface keyword followed by the interface name and optional modifiers.

  For example:

  ```
  public interface MyInterface {
  // interface body
  }
  ```

- **Class or Interface Body:** The body of a class or interface contains fields, methods, constructors, and nested classes or interfaces. It is enclosed in curly braces {}.

  For example:

- **Comments:** Java supports single-line comments (//) and multi-line comments (/* */) for adding explanations or documentation to the code.

How to Run Java Program

- Write Your Java Program: Create a Java source file with a .java extension.
- For example, let's say you have a simple program called HelloWorld.java

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

- ❑ **Compile Your Java Program:**

Open a command prompt and navigate to the directory containing your Java source file.

Use the javac command to compile your program.

```
javac HelloWorld.java
```

- ❑ **Run Your Java Program:** Use the java command to run your compiled Java program.

```
java HelloWorld
```

This will execute your HelloWorld class, and you should see the output Hello, world! printed to the console.

Class in Java

Class is a template or blueprint from which objects are created.

A class in java can contain:

- **data member**
- **method**
- **constructor**
- **block**
- **class and interface**

Syntax to declare a class

```
class <class_name>
{
    data member;
    method;
}
```

Object in Java

- **Object is an instance of a class.**
- Class is a template or blueprint from which objects are created.

An object has three characteristics:

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

Constructor in Java

- **Constructor in java** is a *special type of method* that is used to initialize the object.
- Java constructor is *invoked at the time of object creation*.
- It constructs the values i.e. provides data for the object that is why it is known as constructor.

Rules for creating java constructor

There are basically two rules defined for the constructor.

- ❑ Constructor name must be same as its class name
- ❑ Constructor must have no explicit return type

Types of java constructors

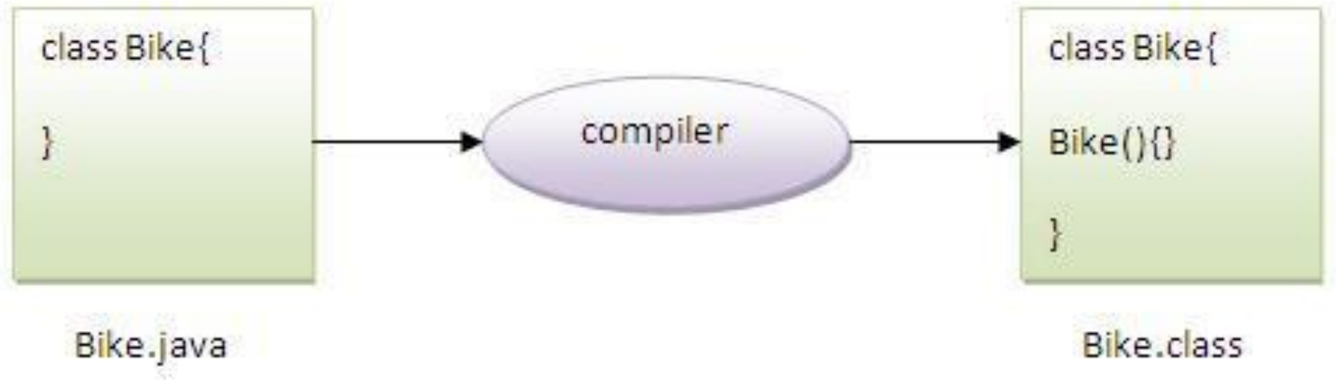
There are two types of constructors:

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

Example of 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.

```
class Bike1{
Bike1(){System.out.println("Bike is created");}
public static void main(String args[]){
Bike1 b=new Bike1();
}}
```



Rule: If there is no constructor in a class, compiler automatically creates a default constructor.

Example of parameterized constructor

```
class Student4{
    int id;
    String name;

    Student4(int i,String n){
    id = i;
    name = n;
    }
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
    Student4 s1 = new Student4(111,"Karan");
    Student4 s2 = new Student4(222,"Aryan");
    s1.display();
    s2.display();
   }
}
```

Constructor Overloading in Java

- Constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter lists.
- The compiler differentiates these constructors by taking into account the number of parameters in the list and their type.

```
class Student5{
    int id;
    String name;
    int age;
    Student5(int i,String n){
    id = i;
    name = n;
    }
    Student5(int i,String n,int a){
    id = i;
    name = n;
    age=a;
    }
    void display(){System.out.println(id+" "+name+" "+age);}

    public static void main(String args[]){
    Student5 s1 = new Student5(111,"Karan");
    Student5 s2 = new Student5(222,"Aryan",25);
    s1.display();
    s2.display();
   }
}
```

Java Copy Constructor

- There is no copy constructor in java. But, we can copy the values of 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
- By assigning the values of one object into another

```
class Student6{
    int id;
    String name;
    Student6(int i,String n){
    id = i;
    name = n;
    }
    Student6(Student6 s){
    id = s.id;
    name =s.name;
    }
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
    Student6 s1 = new Student6(111,"Karan");
    Student6 s2 = new Student6(s1);
    s1.display();
    s2.display();
   }
}
```

Methods in Java

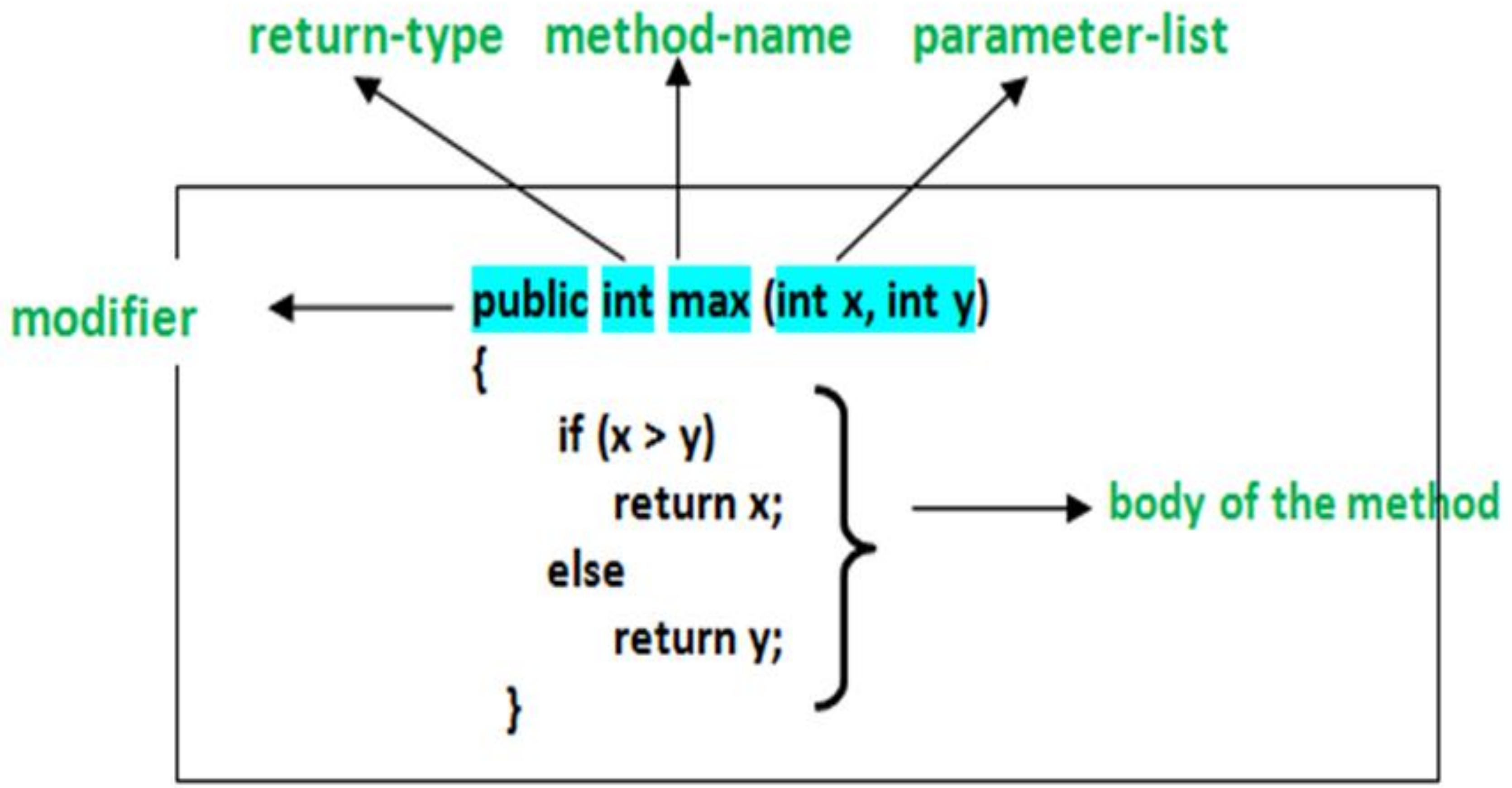
Methods of Java is a collection of statements that perform some specific task and return the result to the caller.

1. A method is like a function i.e. used to expose the behavior of an object.
2. It is a set of codes that perform a particular task.

Syntax of Method

```
<access_modifier> <return_type> <method_name>( list_of_parameters)
{
   //body
}
```

Method Declaration



```
public int max (int x, int y)
{
    if (x > y)
        return x;
    else
        return y;
}
```

Access Specifiers in Java

In Java, **access specifiers** (also called **access modifiers**) define the **scope** and **visibility** of classes, methods, and variables.

Modifier	**Within Same Class**	**Within Same Package**	**Subclass in Another Package**	**Other Packages**
public	✓ Yes	✓ Yes	✓ Yes	✓ Yes
protected	✓ Yes	✓ Yes	✓ Yes	✗ No
(default)	✓ Yes	✓ Yes	✗ No	✗ No
private	✓ Yes	✗ No	✗ No	✗ No

Access Specifiers in Java

- public :Accessible from anywhere in the program.
- Protected : Accessible Within the same package. In subclasses (even if in different packages).
- (default) (no modifier specified)Also called package-private : Accessible only within the same package.
- Private : Accessible only within the same class. Not visible to other classes, even in the same package.

Java Data Types

In Java, data types define the kind of data a variable can hold.

They are mainly divided into primitive and non-primitive (reference) types.

1. Primitive Data Types

There are **8 primitive types** in Java — predefined by the language and stored directly in memory.

Type	Size	Default Value	Example	Description
byte	1 byte	0	byte a = 10;	Small integer (-128 to 127)
short	2 bytes	0	short s = 1000;	Medium integer (-32,768 to 32,767)
int	4 bytes	0	int i = 100000;	Default integer type
long	8 bytes	0L	long l = 100000L;	Large integer
float	4 bytes	0.0f	float f = 3.14f;	Single-precision decimal
double	8 bytes	0.0d	double d = 3.14159;	Double-precision decimal
char	2 bytes	'\u0000'	char c = 'A';	Single character (Unicode)
boolean	1 bit*	false	boolean b = true;	True/false values

Java Data Types

2. Non-Primitive (Reference) Data Types

These are created by the programmer or provided in Java libraries.

Examples

- ❑ String

```
String name = "Java";
```

- ❑ Arrays

```
int[] numbers = {1, 2, 3};
```

- ❑ Classes & Objects

```
class Student { String name; }
```

- ❑ Interfaces

Java Variable Types

Variable is name of reserved area allocated in memory.

There are three kinds of variables in Java:

- ❑Local variables
- ❑Instance variables
- ❑Class/static variables

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

Local variables :

- ❑ Local variables are declared in methods, constructors, or blocks.
- ❑ Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor or block.
- ❑ Access modifiers cannot be used for local variables.
- ❑ Local variables are visible only within the declared method, constructor or block.
- ❑ Local variables are implemented at stack level internally.
- ❑ There is no default value for local variables so local variables should be declared and an initial value should be assigned before the first use.

Instance variables :

- Instance variables are declared in a class, but outside a method, constructor or any block.
- When a space is allocated for an object in the heap a slot for each instance variable value is created.
- Instance variables are created when an object is created with the use of the key word 'new' and destroyed when the object is destroyed.
- Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object’s state that must be present through out the class.

Class/static variables :

- ❑Class variables also known as static variables are declared with the *static* keyword in a class, but outside a method, constructor or a block.
- ❑There would only be one copy of each class variable per class, regardless of how many objects are created from it.
- ❑Static variables are stored in static memory. It is rare to use static variables other than declared final and used as either public or private constants.

Static Keyword in Java

- The static keyword in Java is mainly used for memory management.
- The static keyword in Java is used to share the same variable or method of a given class.
- We can apply static keywords with variables, methods, blocks, and nested classes.
- The static keyword belongs to the class than an instance of the class.

The *static* keyword is a non-access modifier in Java that is applicable for the following:

1. Blocks
2. Variables
3. Methods
4. Classes

Note: *To create a static member(block, variable, method, nested class), precede its declaration with the keyword static.*

Characteristics of static keyword:

- ❑ **Shared memory allocation:** Static variables and methods are allocated memory space only once during the execution of the program. This memory space is shared among all instances of the class, which makes static members useful for maintaining global state or shared functionality.
- ❑ **Accessible without object instantiation:** Static members can be accessed without the need to create an instance of the class. This makes them useful for providing utility functions and constants that can be used across the entire program.

- **Associated with class, not objects:** Static members are associated with the class, not with individual objects. This means that changes to a static member are reflected in all instances of the class, and that you can access static members using the class name rather than an object reference.
- **Cannot access non-static members:** Static methods and variables cannot access non-static members of a class, as they are not associated with any particular instance of the class.

- **Can be overloaded, but not overridden**: Static methods can be overloaded, which means that you can define multiple methods with the same name but different parameters. However, they cannot be overridden, as they are associated with the class rather than with a particular instance of the class.

```
class Test
{
    // static method
    static void m1()
    {
        System.out.println("from m1");
    }
    public static void main(String[] args)
    {
          // calling m1 without creating
          // any object of class Test
           m1();
    }
}
```

Static blocks

- If you need to do the computation in order to initialize your static variables, you can declare a static block that gets executed exactly once, when the class is first loaded.

```
class Test
{
   // static variable
   static int a = 10;
   static int b;
   // static block
   static {
      System.out.println("Static block initialized.");
      b = a * 4;
   }
    public static void main(String[] args)
   {
     System.out.println("from main");
     System.out.println("Value of a : "+a);
     System.out.println("Value of b : "+b);
   }
}
```

Static variables

- ❑When a variable is declared as static, then a single copy of the variable is created and shared among all objects at the class level.
- ❑Static variables are, essentially, global variables.
- ❑All instances of the class share the same static variable.

Important points for static variables:

- We can create static variables at the class level only.
- static block and static variables are executed in the order they are present in a program.

Static methods

When a method is declared with the static keyword, it is known as the static method. The most common example of a static method is the main( ) method.

Methods declared as static have several restrictions:

- ❑They can only directly call other static methods.
- ❑They can only directly access static data.

Java static method

If you apply static keyword with any method, it is known as static method.

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

Static Classes

- A class can be made static only if it is a nested class.
- We cannot declare a top-level class with a static modifier but can declare nested classes as static.
- Such types of classes are called Nested static classes.

```
class OuterClass
{
      private static String msg = “KIET Group of Institutions";
       public static class NestedStaticClass
        {
           public void printMessage()
           {
                      System.out.println("Message " + msg);
           }
         }
}
```

```
class MyMain {
            public static void main(String args[])
            {
OuterClass.NestedStaticClass printer= new
OuterClass.NestedStaticClass();
                        printer.printMessage();
            }
}
```

Final Keyword In Java

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.

Final Variable →	To Create constant variable
Final Methods →	Prevent Method Overriding
Final Classes →	Prevent Inheritance

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

```
class Bike9{
 final int speedlimit=90;//final variable
 void run(){
  speedlimit=400;
 }
 public static void main(String args[]){
 Bike9 obj=new  Bike9();
 obj.run();
 }
}//end of class
```

Output: Compile Time Error

Java final method

If you make any method as final, you cannot override it.

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

class Honda extends Bike{
  void run(){System.out.println("running safely with 100kmph
   ");}

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

Output: Compile Time Error

Java final class

If you make any class as final, you cannot extend it.

```
final class Bike{}

class Honda1 extends Bike{
  void run(){System.out.println("running safely with 100k
    mph");}

  public static void main(String args[]){
  Honda1 honda= new Honda();
  honda.run();
  }
}
```

Output:Compile Time Error

- ❑ A final variable that is not initialized at the time of declaration is known as blank final variable.
- ❑ If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful.

Can we initialize blank final variable?

Yes, but only in constructor. For example:

```
class Bike10{
  final int speedlimit;//blank final variable

  Bike10(){
  speedlimit=70;
  System.out.println(speedlimit);
  }

  public static void main(String args[]){
    new Bike10();
 }
}
```

Abstraction in Java

- **Abstraction in Java** means showing **only the essential features** of an object and hiding the background implementation details.
- It focuses on **what** an object does, not **how** it does it.

Ways to Achieve Abstraction in Java

- **Abstract classes** (0–100% abstraction)
- **Interfaces** (100% abstraction before Java 8, can have default & static methods after Java 8)

Abstraction with Abstract Class

An abstract class cannot be instantiated. It can have abstract methods (without a body) and non-abstract methods (with a body).

Abstraction with Interface

- ❑Defines only method signatures, no method body (until Java 8+ allows default/static methods).
- ❑Achieves **full abstraction** by separating the definition from the implementation.

- Rule 1: If you are extending any abstract class that have abstract method, you must either provide the implementation of the method or make this class abstract.
- Rule 2: If there is any abstract method in a class, that class must be abstract.

```
// Abstract Class (Abstraction)
abstract class Student {
    String name;
    int rollNo;
    // Constructor
    Student(String name, int rollNo) {
        this.name = name;
        this.rollNo = rollNo;
    }
    // Abstract methods (must be implemented differently)
    abstract void attendClass();
    abstract void giveExam();
    // Concrete method (common functionality for all students)
    void register() {
        System.out.println(name + " (Roll No: " + rollNo + ") is registered
successfully.");
    }
}
```

```
// School Student (different implementation)
class SchoolStudent extends Student {
    SchoolStudent(String name, int rollNo) {
        super(name, rollNo);
    }

    @Override
    void attendClass() {
        System.out.println(name + " attends school in a physical
classroom.");
    }

    @Override
    void giveExam() {
        System.out.println(name + " writes exams on paper in
school.");
    }
}
```

```
// College Student (different implementation)
class CollegeStudent extends Student {
    CollegeStudent(String name, int rollNo) {
        super(name, rollNo);
    }

    @Override
    void attendClass() {
        System.out.println(name + " attends lectures in college 
auditorium.");
    }

    @Override
    void giveExam() {
        System.out.println(name + " gives exams online via 
university portal.");
    }
}
```

```
public class AbstractionDemo {
    public static void main(String[] args) {
        Student s1 = new SchoolStudent("Rahul", 101);
        Student s2 = new CollegeStudent("Anita", 202);

        s1.register();   // common method
        s1.attendClass();
        s1.giveExam();

        System.out.println("-----------------------");

        s2.register();   // common method
        s2.attendClass();
        s2.giveExam();
    }
}
```

Interface in Java

In Java, an **interface** is like a **contract** that defines *what* a class should do, but not *how* it should do it. It contains **abstract methods** (methods without a body), and any class that implements the interface must provide the method bodies.

Methods

- ❑ By default: public abstract (even if not written explicitly).
- ❑ From Java 8: Can have default and static methods with body.
- ❑ From Java 9: Can have private methods inside interfaces.

Variables

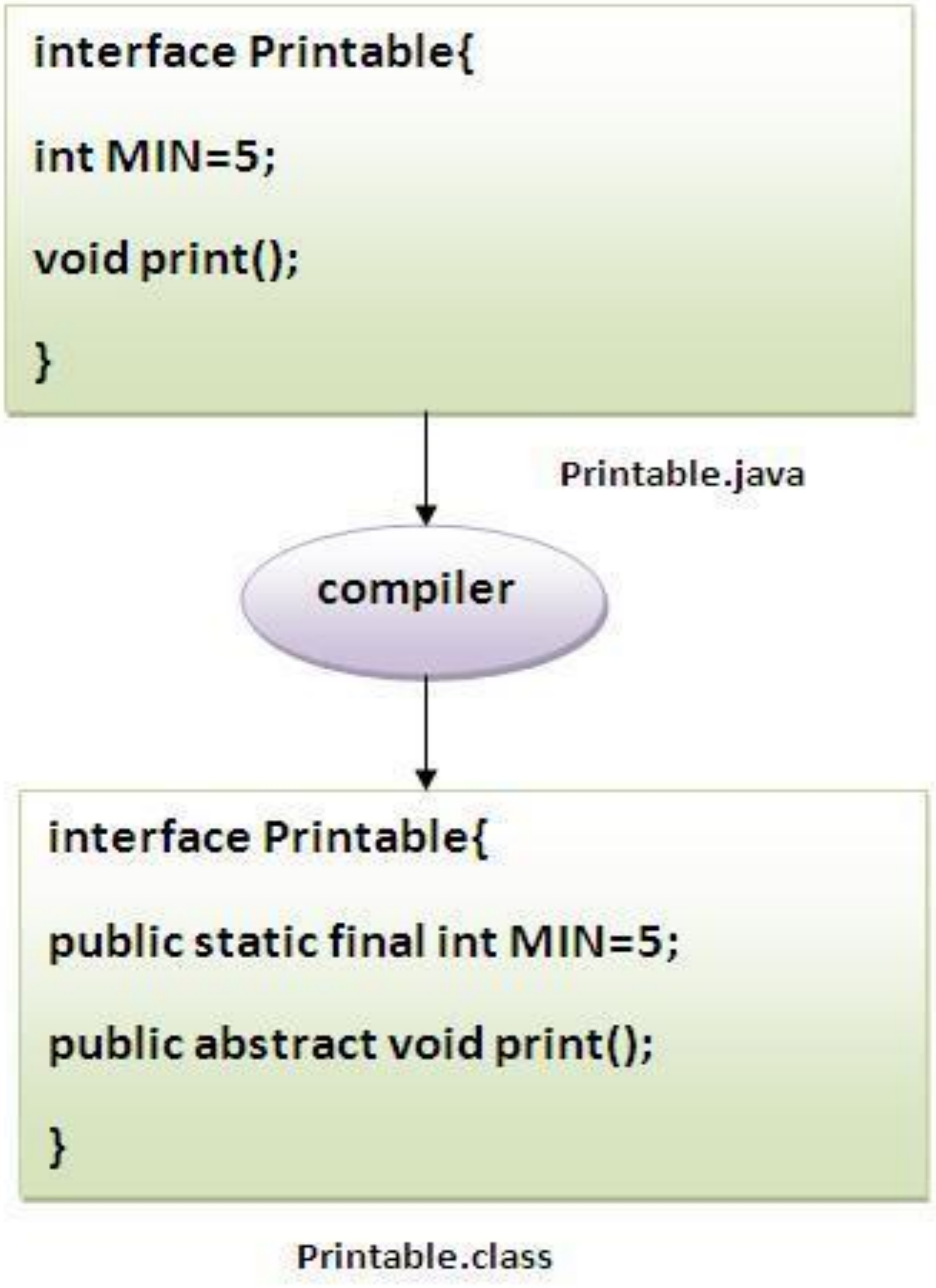
By default: public static final (constants).

Multiple Inheritance

A class can implement multiple interfaces (since there’s no ambiguity of state like in multiple class inheritance).

The java compiler adds public and abstract keywords before the interface method and public, static and final keywords before data members.

In other words, Interface fields are public, static and final by default, and methods are public and abstract.



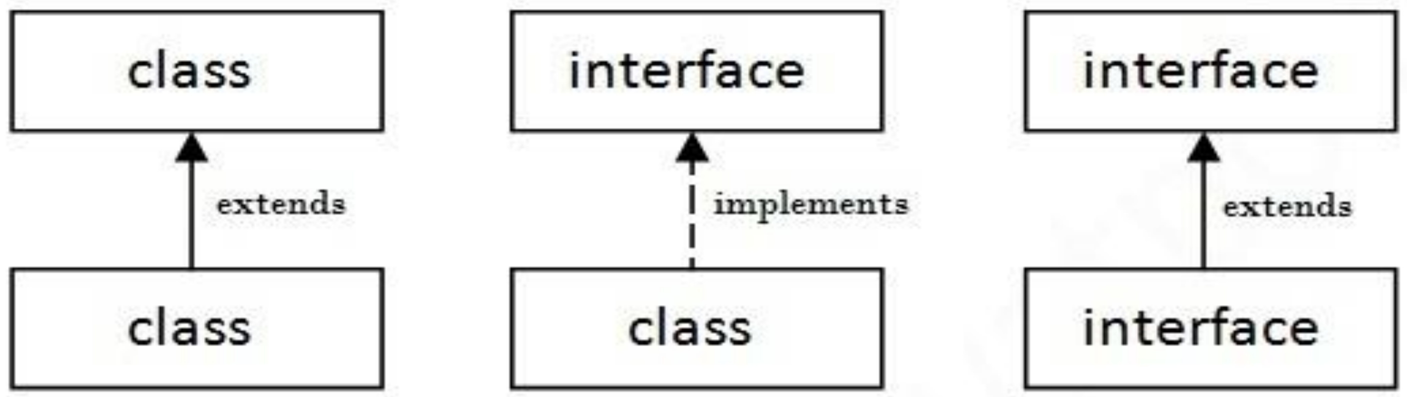
```
interface Printable{

int MIN=5;

void print();

}
```

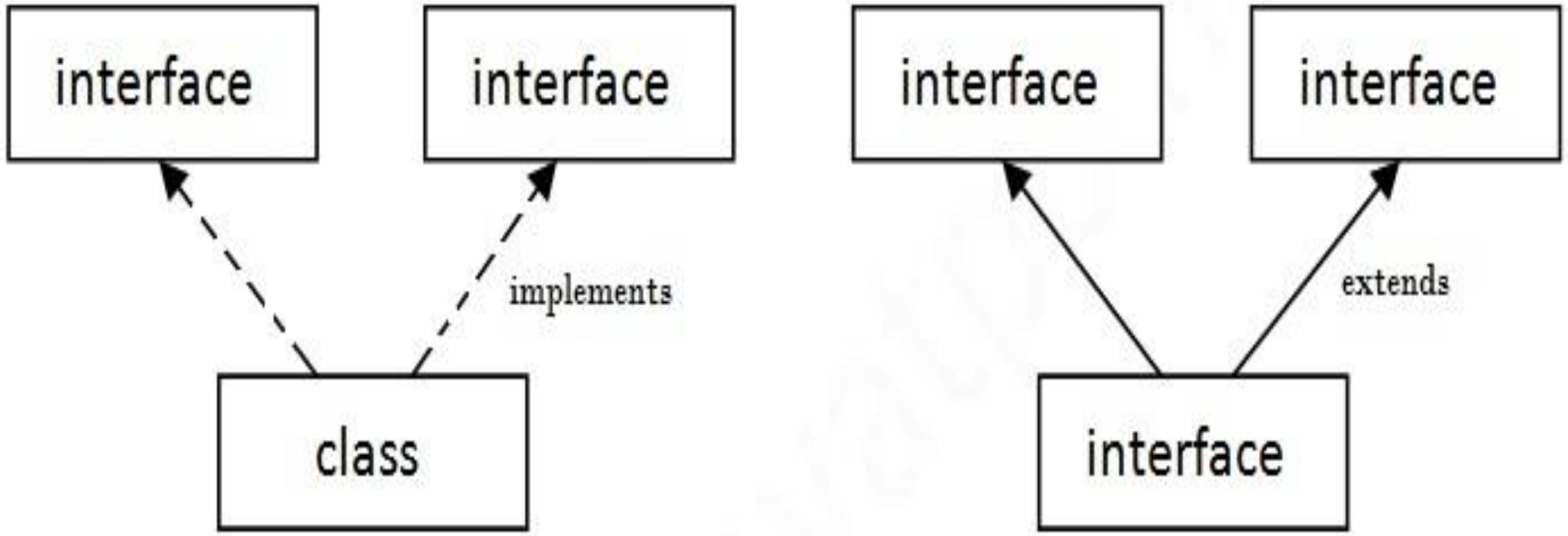
```
interface Printable{

public static final int MIN=5;

public abstract void print();

}
```

```
interface printable{
void print();
}

class A6 implements printable{
public void print(){System.out.println("Hello");}

public static void main(String args[]){
A6 obj = new A6();
obj.print();
 }
}
```

Multiple inheritance in Java by interface

- If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.





Multiple Inheritance in Java

```
interface Printable{
void print();
}

interface Showable{
void show();
}

class A7 implements Printable,Showable{

public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
A7 obj = new A7();
obj.print();
obj.show();
 }
}
```

Feature	Abstract Class	Interface
Methods	Can have **abstract** and **concrete** methods.	Can have only **abstract methods** (till Java 7), plus **default**, **static**, and **private methods** (Java 8+).
Variables	Can have instance variables, static variables, final variables.	Can have only **public static final constants** (implicitly).
Access Modifiers	Abstract methods can have any access modifier (public, protected, etc.).	All methods are **public** by default.
Constructors	Can have constructors.	Cannot have constructors.
Multiple Inheritance	A class can extend **only one** abstract class (because Java does not allow multiple inheritance).	A class can implement **multiple interfaces**.

Encapsulation

- **Encapsulation** is defined as the wrapping up of data under a single unit.
- It is the mechanism that binds together code and the data it manipulates.
- It is a protective shield that prevents the data from being accessed by the code outside this shield.
- Technically in encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function of its own class in which it is declared.

- Encapsulation can be achieved by Declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables.
- It is more defined with the setter and getter method.

Advantages of Encapsulation

- **Data Hiding:** it is a way of restricting the access of our data members by hiding the implementation details. Encapsulation also provides a way for data hiding.
- **Reusability:** Encapsulation also improves the re-usability and is easy to change with new requirements.
- **Testing code is easy:** Encapsulated code is easy to test for unit testing.
- **Freedom to programmer in implementing the details of the system.**

Disadvantages of Encapsulation in Java

- Can lead to increased complexity, especially if not used properly.
- Can make it more difficult to understand how the system works.
- May limit the flexibility of the implementation.

```
class Student {
    // Private fields (Encapsulation)
    public int id;
    public String name;
    private double grade;

    // Constructor
    public Student(int id, String name, double grade)
{
        this.id = id;
        this.name = name;
        this.grade = grade;
    }
```

```
 // Getters and Setters (Controlled access)

    public double getGrade() { return grade; }
    public void setGrade(double grade) {
this.grade = grade; }

    // Method to display student info
    public void displayInfo() {
        System.out.println("ID: " + id + ", Name: " +
name + ", Grade: " + grade);
    }
}
```

Inheritance in Java

- ❑ Inheritance is an important pillar of OOP(Object-Oriented Programming).
- ❑ It is the mechanism in Java by which one class is allowed to inherit the features(fields and methods) of another class.

Why use inheritance in java

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

Syntax of Java Inheritance

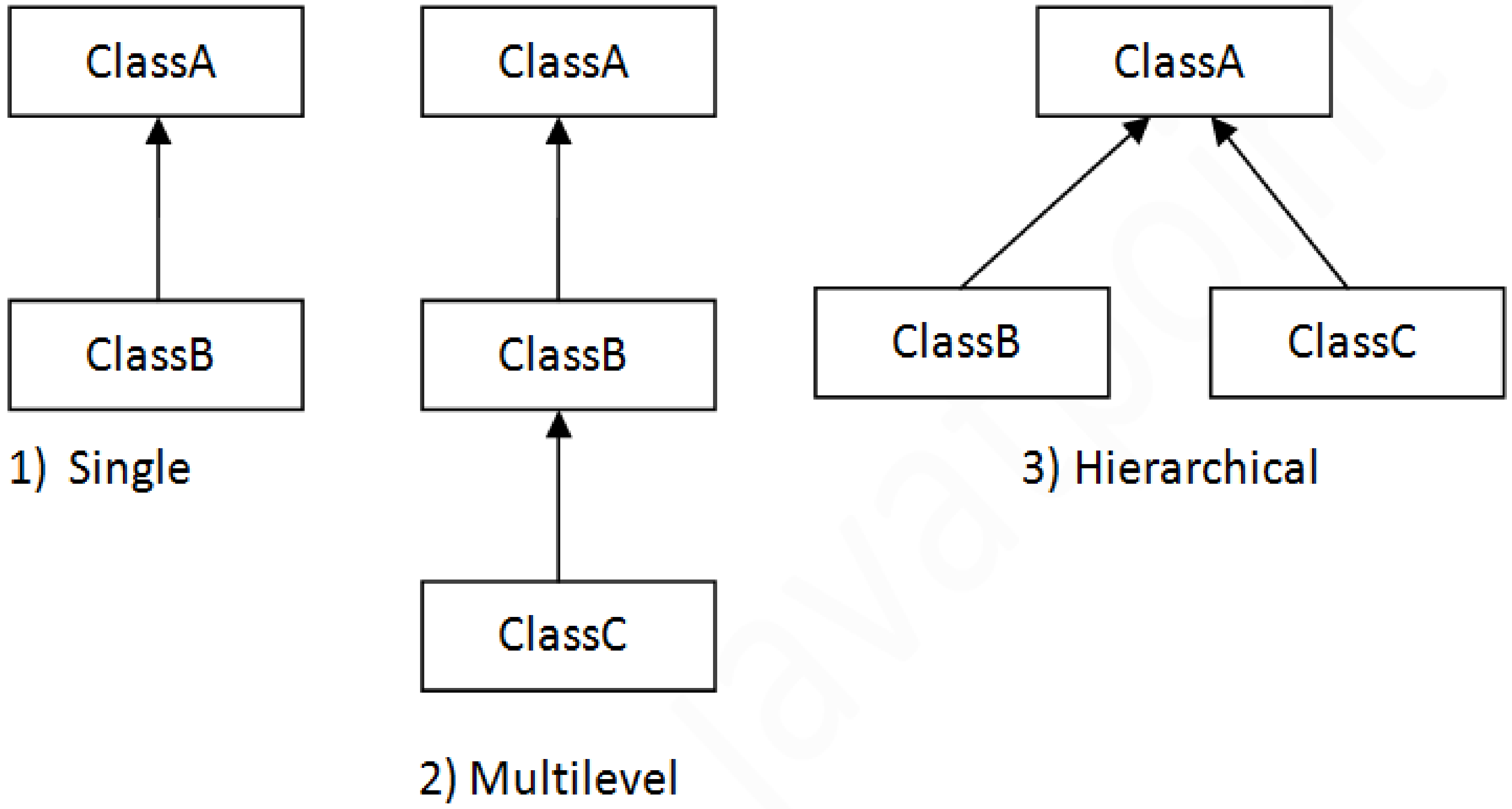
```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

The **extends keyword** indicates that you are making a new class that derives from an existing class.

```java
// Parent class
class Person {
    protected String address;

    public Person(String address) {
        this.address = address;
    }

    public void showAddress() {
        System.out.println("Address: " + address);
    }
}
```

```
// Student inherits Person
class CollegeStudent extends Person {
    private String course;

    public CollegeStudent(String address, String course) {
        super(address); // calling parent constructor
        this.course = course;
    }

    public void showCourse() {
        System.out.println("Course: " + course);
    }
}
```

Types of inheritance in 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 same method and you call it from child class object, there will be ambiguity to call method of A or B class.
- Since compile time errors are better than runtime errors.

```
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 would be invoked?
}
}
```

Compile Time Error

Polymorphism

- Polymorphism is considered one of the important features of Object-Oriented Programming.
- Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations.
- The word “poly” means many and “morphs” means forms, So it means many forms.

Types of Java Polymorphism

In Java Polymorphism is mainly divided into two types

- ❑Compile-time Polymorphism
- ❑Runtime Polymorphism

Compile-Time Polymorphism in Java

It is also known as static polymorphism. This type of polymorphism is achieved by function overloading or operator overloading.

Note: *Java doesn’t support the Operator Overloading.*

Method Overloading

When there are multiple functions with the same name but different parameters then these functions are said to be overloaded.

Functions can be overloaded by changes in the number of arguments or/and a change in the type of arguments.

```
class StudentPoly {
   // Method Overloading (Compile-time Polymorphism)
    public void study() {
        System.out.println("Student is studying");
    }

    public void study(String subject) {
        System.out.println("Student is studying " + 
subject);
    }
}
```

Runtime Polymorphism

It is also known as Dynamic Method Dispatch. It is a process in which a function call to the overridden method is resolved at Runtime. This type of polymorphism is achieved by Method Overriding.

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 is used to provide specific implementation of a method that is already provided by its super class.
- Method overriding is used for runtime polymorphism.

// Runtime Polymorphism

```
class Undergraduate extends StudentPoly {
    @Override
    public void study() {
        System.out.println("Undergraduate student is studying
with notes");
    }
}

class Postgraduate extends StudentPoly {
    @Override
    public void study() {
        System.out.println("Postgraduate student is researching
with journals");
    }
}
```

Rules for Java Method Overriding

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

```
class Vehicle{
 void run(){System.out.println("Vehicle is runnin
   g");}
}
class Bike extends Vehicle{  void run(){System.o
   ut.println(“Bike  is running");}


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

Exception Handling

- ❑Exception Handling in Java is one of the effective means to handle runtime errors so that the regular flow of the application can be preserved.
- ❑In Java, an **Exception** is an event that disrupts the normal flow of a program’s execution. It usually occurs when something unexpected happens, such as dividing by zero, accessing an invalid array index, or trying to open a file that doesn’t exist.

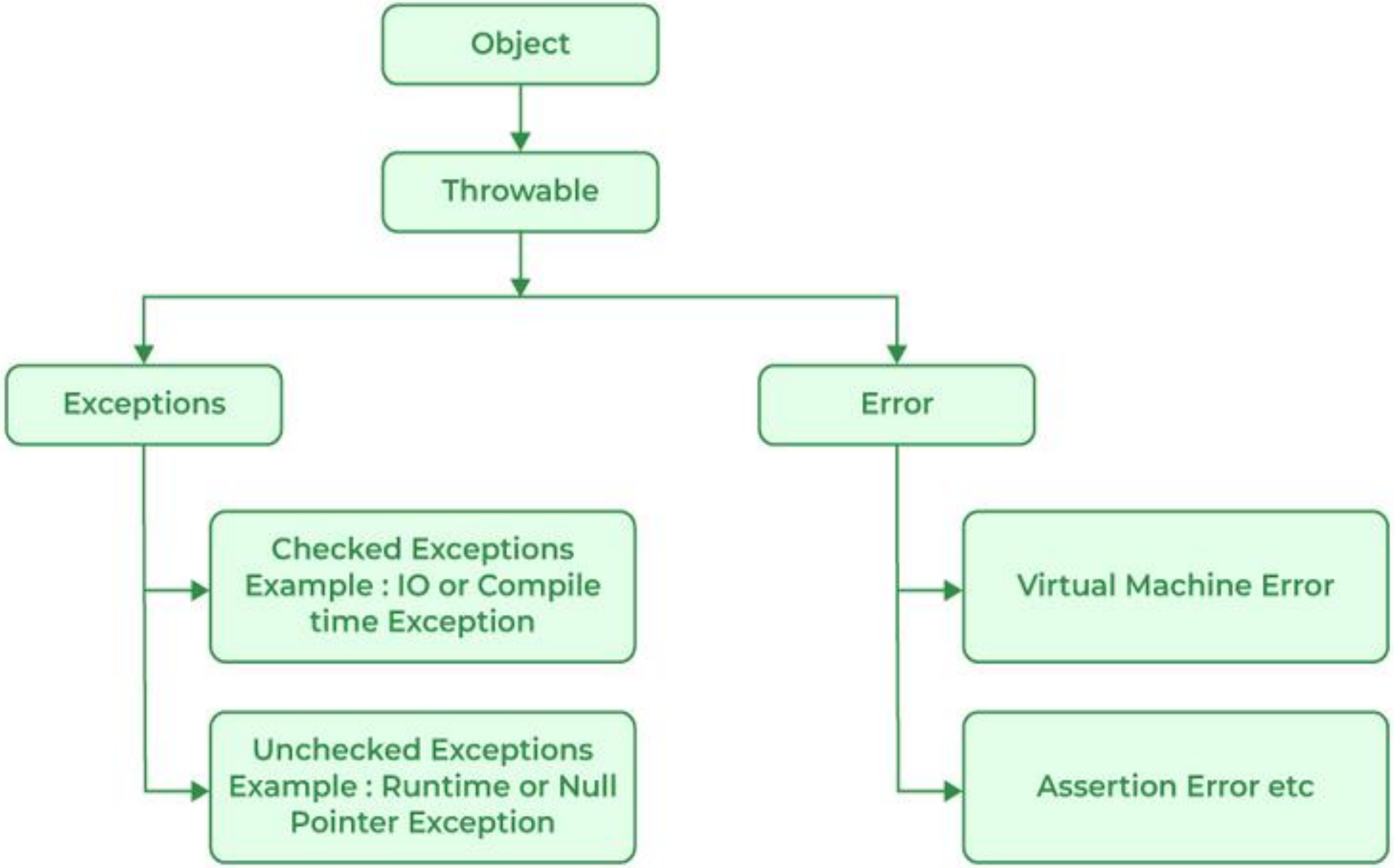
Major reasons why an exception Occurs

- ❑Invalid user input
- ❑Device failure
- ❑Loss of network connection
- ❑Physical limitations (out-of-disk memory)
- ❑Code errors
- ❑Opening an unavailable file

Errors

- Errors represent irrecoverable conditions such as Java virtual machine (JVM) running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc.
- Errors are usually beyond the control of the programmer, and we should not try to handle errors.

Exception Hierarchy

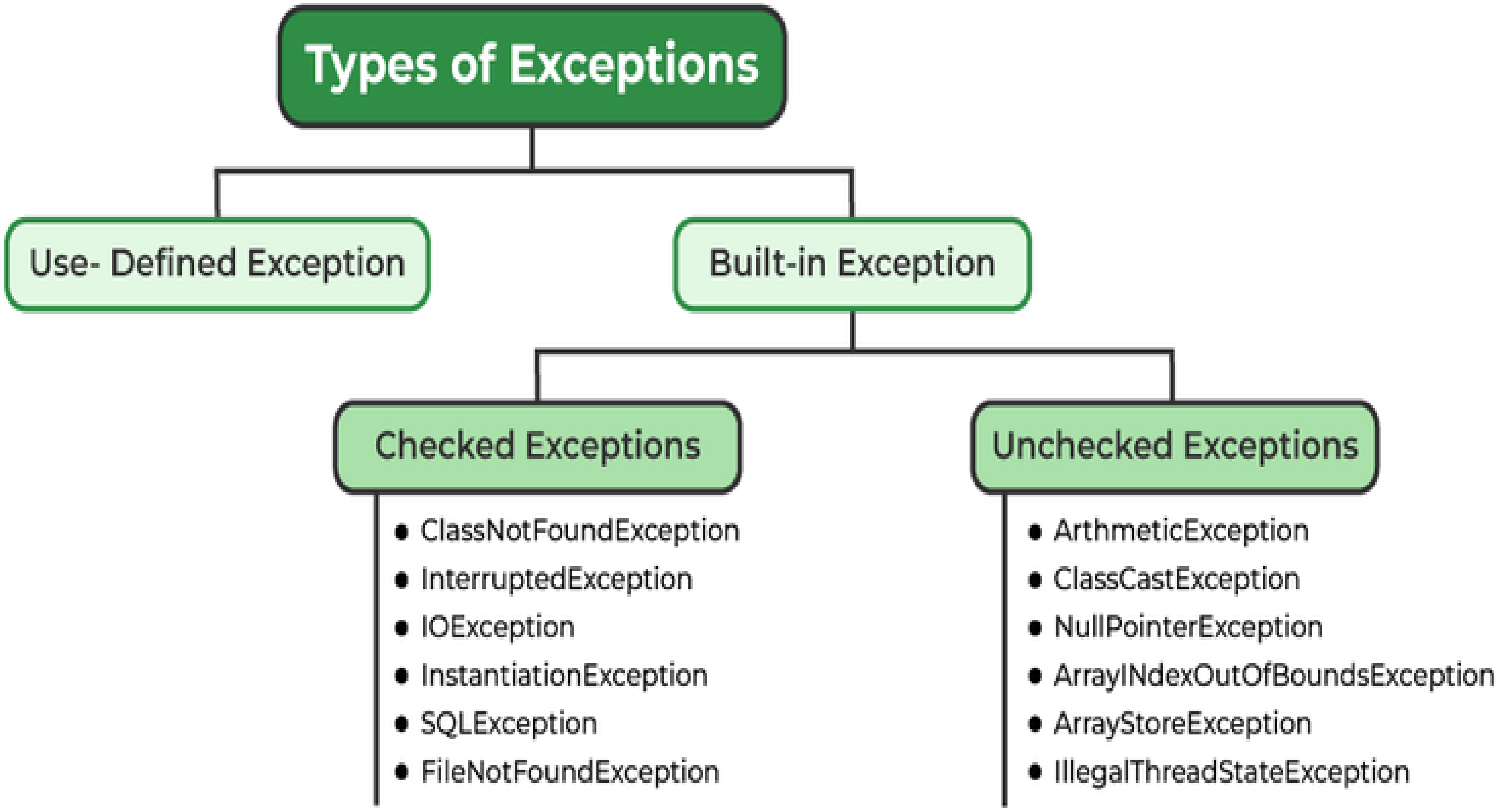


Object
Throwable
Exceptions
Error
Checked Exceptions Example : IO or Compile time Exception
Unchecked Exceptions Example : Runtime or Null Pointer Exception
Virtual Machine Error
Assertion Error etc

Difference between Error and Exception

- Error: An Error indicates a serious problem that a reasonable application should not try to catch.
- Exception: Exception indicates conditions that a reasonable application might try to catch.

Types of Exceptions



Built-in Exceptions

❑Checked Exceptions

Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler.

❑Unchecked Exceptions:

The compiler will not check these exceptions at compile time.

if a program throws an unchecked exception, and even if we didn’t handle or declare it, the program would not give a compilation error.

User-Defined Exceptions:

Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, users can also create exceptions, which are called ‘user-defined Exceptions’.

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.

Flow control in try catch finally in Java

1. **Control flow in try-catch clause OR try-catch-finally clause**
    - ❑**Case 1:** Exception occurs in try block and handled in catch block
    - ❑**Case 2:** Exception occurs in try-block is not handled in catch block
    - ❑**Case 3:** Exception doesn’t occur in try-block
2. **try-finally clause**
    - ❑**Case 1:** Exception occurs in try block
    - ❑**Case 2:** Exception doesn’t occur in try-block

Exception occurs in try block and handled in catch block

If a statement in try block raised an exception, then the rest of the try block doesn’t execute and control passes to the **corresponding** catch block.

After executing the catch block, the control will be transferred to finally block(if present) and then the rest program will be executed.

Exception occurred in try-block is not handled in catch block

In this case, the default handling mechanism is followed.

If finally block is present, it will be executed followed by the default handling mechanism.

Exception doesn’t occur in try-block:

In this case catch block never runs as they are only meant to be run when an exception occurs. finally block(if present) will be executed followed by rest of the program.

```
class A
{
    public static void main (String[] args)
    {
    try
    {
      String str = "123";
       int num = Integer.parseInt(str);
       System.out.println("try block fully executed");
     }
    catch(NumberFormatException ex)
    {
     System.out.println("catch block executed...");
     }
     finally
    {
        System.out.println("finally block executed");
    }
   System.out.println("Outside try-catch-finally clause");
    }
}
```

Control flow in try-finally

In this case, no matter whether an exception occurs in try-block or not finally will always be executed. But control flow will depend on whether an exception has occurred in the try block or not.

- Exception raised: If an exception has occurred in the try block then the control flow will be finally block followed by the default exception handling mechanism.

```
class A
{
    public static void main (String[] args)
    {
       int[] arr = new int[4];
        try
        {
            int i = arr[4];
            System.out.println("Inside try block");
        }
        finally
        {
            System.out.println("finally block executed");
        }
        // rest program will not execute
        System.out.println("Outside try-finally clause");
    }
}
```

❑ Exception not raised:

If an exception does not occur in the try block then the control flow will be finally block followed by the rest of the program

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.

```
public class MultipleCatchBlock1 {
  public static void main(String[] args) {
           try{
             int a[]=new int[5];
             a[5]=30/0;
            }
            catch(ArithmeticException e)
              {
               System.out.println("Arithmetic Exception occurs");
              }
            catch(ArrayIndexOutOfBoundsException e)
              {
               System.out.println("ArrayIndexOutOfBounds Exception occurs");
              }
            catch(Exception e)
              {
               System.out.println("Parent Exception occurs");
              }
            System.out.println("rest of the code");
   }
}
```

```java
public class MyMain {
public static void main(String[] args) {
int a[]=new int[4];
try
{
a[0]=12/0;
System.out.println(a[6]);
}
catch(ArithmeticException e)
{
System.out.println("Aritmetic Exception"+e.getMessage());
}
catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("Index out of bound"+e.getMessage());
}
catch(Exception e)
{
System.out.println(e.getMessage());
}
finally
{
System.out.println("Finally block executed");
}
System.out.println("outside Finally block executed");
}
}
```

throw keyword

- The Java throw keyword is used to throw an exception explicitly.
- We specify the **exception** object which is to be thrown. The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.
- We can throw either **checked** or **unchecked** exceptions in Java by throw keyword. It is mainly used to throw a custom exception.

syntax of the Java throw keyword

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

Example of throw IOException.

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

Example of Java throw keyword

```
public class Main {
    public static void main(String[] args) {
        int dividend = 10;
        int divisor = 0;

        if (divisor == 0) {
            throw new ArithmeticException("Cannot divide by zero");
        }
        int result = dividend / divisor;
        System.out.println("Result: " + result);
    }
}
```

Throwing Unchecked Exception

```
public class TestThrow1 {
    public static void validate(int age) {
        if(age<18) {
            //throw Arithmetic exception if not eligible to vote
  throw new ArithmeticException("Person is not eligible to vote");

        }
        else {
            System.out.println("Person is eligible to vote!!");
        }
    }
```

```
 public static void main(String args[]){
     //calling the function
     validate(13);
     System.out.println("rest of the code...");
  }
}
```

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

Java throws keyword

- The **Java throws keyword** is used to declare an exception.
- It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.
- if there is a chance of raising an exception then the compiler always warns us about it and compulsorily we should handle that checked exception, Otherwise, we will get compile time error saying unreported exception “xyz” must be caught or declared to be thrown.

Syntax of java throws

```
return_type method_name() throws excepti
   on_class_name
{
//method code
}
```

Important Points to Remember about throws Keyword

- throws keyword is required only for checked exceptions and usage of the throws keyword for unchecked exceptions is meaningless.
- throws keyword is required only to convince the compiler and usage of the throws keyword does not prevent abnormal termination of the program.
- With the help of the throws keyword, we can provide information to the caller of the method about the exception.

Steps to Create a User-Defined Exception

- ❑Create a class that extends Exception (for checked exceptions) or RuntimeException (for unchecked exceptions).
- ❑Provide a constructor to pass the error message to the parent class.
- ❑Throw the custom exception where needed.
- ❑Handle it using try-catch.

Example 1: User-Defined Checked Exception

```
// Step 1: Define custom exception
public class InvalidAgeException extends
Exception {
    public InvalidAgeException(String message)  {
        super(message);
    }
}
```

// Step 2: Use custom exception

```
public class UserDefineException {
   public void validate(int age) throws InvalidAgeException{
      if(age<18)
         throw new InvalidAgeException("Age less than 18, Not
valid to vote");
      else
         System.out.println("Eligible to Vote");


   }
}
```

```
public class MyException  {
    public static void main(String[] args) {
        UserDefineException obj=new UserDefineException();
        try {
            obj.validate(12);
            obj.validate(19);
        }
        catch (InvalidAgeException e)
        {
            System.out.println(e.getMessage());
        }
    }
}
```

Example 2: User-Defined Checked Exception

// Step 1: Define custom exception

```
public class InsufficientBalanceException extends
RuntimeException{
    public InsufficientBalanceException(String message)
    {
        super(message);
    }
}
```

// Step 2: Use custom exception

```
public class Bank {
    private int balance=10000;
    public void withdraw(int amount)
    {
        if(amount>balance)
        {
            throw new InsufficientBalanceException("Not Enough 
Balance");
        }
        else {
            balance-=amount;
            System.out.println("Withdrawal Successfull , Remaining Balance 
"+balance);
        }
    }
}
```

```
public class MyBank {
    public static void main(String[] args) {
        Bank b=new Bank();
        try
        {
            b.withdraw(5000);
        }
        catch (InsufficientBalanceException e)
        {
            System.out.println(e.getMessage());
        }
    }
}
```