Java Core Concepts

Java Platform

JVM Java Virtual Machine

- It runs Java programs
- Converts bytecode to machine code
- Handles memory and garbage collection

JRE Java Runtime Environment

- Provides libraries to run Java Applications
- It includes the JVM and standard runtime libraries.

JDK Java Development Kit

- It includes the JRE along with development tools such as the Java compiler.
- Needed to write and develop Java programs

Data Types

Primitive Data Types

Primitive data types store simple and single values. They have a fixed size and are faster to use.

Examples:
int a = 10;
float b = 5.5f;
char c = 'A';

Non-Primitive Data Types

Non-primitive data types store multiple values or objects. Their size is not fixed and they include classes, arrays, and strings.

Examples:
String name = "Java";
int[] arr = {1,2,3};

Class and Object

Class

- A blueprint
- Defines variables and methods

```
class Car {
   String color;
   void drive() {
      System.out.println("Car is moving");
   }
}
```

Object

- Real instance of a class

```
Car c = new Car();
c.drive();
```

Encapsulation

Encapsulation is the process of wrapping data and methods into a single unit. It helps in data hiding by using access modifiers like private. Data can be accessed safely using getters and setters.

```
class Student {
   private int marks;

   public int getMarks() {
      return marks;
   }

   public void setMarks(int m) {
```

```
        marks = m;
    }
}
```

Inheritance

Single Inheritance

- One child, one parent

```
class Animal {
    void eat() {}
}

class Dog extends Animal {
    void bark() {}
}
```

Multilevel Inheritance

- Parent → Child → Grandchild

```
class A {}
class B extends A {}
class C extends B {}
```

Hierarchical Inheritance

- One parent, many children

```
class Animal {}
class Dog extends Animal {}
class Cat extends Animal {}
```

Polymorphism

Polymorphism allows objects of derived classes to be treated as objects of their base class but still maintain their unique behaviors.

Compile-Time Polymorphism (Method Overloading)

- Same method name
- Different parameters

```
class Calc {
   int add(int a, int b) {
      return a + b;
   }

   int add(int a, int b, int c) {
      return a + b + c;
   }
}
```

Run-Time Polymorphism (Method Overriding)

- Child class changes parent method

```
class Animal {
   void sound() {
      System.out.println("Animal sound");
   }
}

class Dog extends Animal {
   void sound() {
      System.out.println("Bark");
```

```
    }
}
```

Abstraction

Abstraction is all about creating a simple interface that exposes only relevant and necessary functionalities.

Abstract Class

- Can have abstract and normal methods
- Cannot create object

```
abstract class Shape {
    abstract void draw();
}
```

Interface

An interface is a contract or blueprint that classes adhere to. Unlike abstract classes, interfaces have no implementation details; they only declare method and property signatures.

```
interface Vehicle {
    void start();
}
```

Interface vs Abstract Class

Interfaces support multiple inheritance, whereas abstract classes do not.

Interfaces provide complete abstraction, while abstract classes allow partial implementation.

SOLID Principles

SRP (Single Responsibility Principle)

A class should have only one responsibility or job.

Example:
Student class should not handle database logic.

OCP (Open/Closed Principle)

A class should have only one responsibility or job.

Example:
Add new feature using new class, not by changing old code.

LSP (Liskov Substitution Principle)

Child class should work properly when used as parent

Example:
If Dog replaces Animal, program should not break.

ISP (Interface Segregation Principle)

Don’t force classes to implement unused methods

Example:
Separate Print and Scan interfaces instead of one big interface.

DIP (Dependency Inversion Principle)

Depend on interfaces, not concrete classes

```
interface Engine {
   void start();
}
class Car {
   Engine e;
   Car(Engine e) {
      this.e = e;
   }
}
```

Exception Handling

Exception handling is used to handle errors so the program does not stop suddenly.

Checked Exceptions

Checked exceptions are checked at compile time and must be handled.

Example:

```
try {
   FileReader f = new FileReader("data.txt");
} catch (Exception e) {
   System.out.println("File not found");
}
```

Unchecked Exceptions

Unchecked exceptions occur at runtime and are caused by programming mistakes.

Example:

```
int a = 10 / 0;   // ArithmeticException
```

try – catch – finally

The try block contains risky code, catch handles the error, and finally always executes.

Example:

```
try {
   int x = 10 / 0;
} catch (Exception e) {
   System.out.println("Error occurred");
} finally {
   System.out.println("Program ended");
}
```

Collections Framework

Collections are used to store and manage groups of objects.

List (ArrayList, LinkedList)

List allows duplicate values and maintains insertion order.

Example:

```
ArrayList<Integer> list = new ArrayList<>();

list.add(10);

list.add(10);

list.add(20);
```

Set (HashSet, TreeSet)

Set does not allow duplicate values.

Example:

```
HashSet<Integer> set = new HashSet<>();

set.add(10);

set.add(10);   // duplicate ignored
```

Map (HashMap, TreeMap)

Map stores data in key–value pairs.

Example:

```
HashMap<Integer, String> map = new HashMap<>();

map.put(1, "Java");

map.put(2, "Python");
```

Multithreading

Multithreading allows multiple tasks to run at the same time.

Thread Class

Thread can be created by extending the Thread class and overriding the run() method.

Example:

```
class MyThread extends Thread {
   public void run() {
      System.out.println("Thread running");
   }
}
```

Runnable Interface

Thread can also be created using Runnable interface which is more flexible since it allows multiple inheritance.

Example:

```
class MyTask implements Runnable {
   public void run() {
      System.out.println("Runnable thread");
   }
}
```

Synchronization

Synchronization prevents multiple threads from accessing shared data at the same time.

Example:

```
synchronized void display() {
   System.out.println("Synchronized method");
}
```

Java Memory Management

Java automatically manages memory.

Stack Memory

Stack memory stores method calls and local variables and follows LIFO order, for example local variables inside a method.

Example:

```
void show() {
    int x = 10;   // stored in stack
}
```

Heap Memory

Heap memory stores objects created using the new keyword and is shared among threads.

Example:

```
Student s = new Student();  // object stored in heap
```

Garbage Collection

Unused objects are removed automatically such as when an object reference is set to null.

Example:

```
Student s = new Student();
s = null;   // eligible for garbage collection
```

Java 8 Features

Lambda Expressions

Lambda provides short syntax for methods.

Example:

```
Runnable r = () -> System.out.println("Hello");
```

Stream API

Stream is used to process data from collections.

Example:

```
list.stream().forEach(n -> System.out.println(n));
```

Optional Class

Optional avoids NullPointerException by wrapping values that may or may not be present.

Example:

```
Optional<String> name = Optional.of("Java");
System.out.println(name.get());
```

Important Keywords

static

Belongs to class, not object.

Example:

```
static int count = 0;
```

final

Cannot be changed or inherited.

Example:

```
final int x = 10;
```

this

Refers to current object.

Example:

this.name = name;

super

Refers to parent class.

Example:

super.display();

volatile

Ensures latest value is read from memory in multithreading.

Example:

volatile int flag;

transient

Prevents variable from serialization.

Example:

transient int password;