JAVA(Beginner)

10 important Questions

1. What is the difference between JDK, JRE, and JVM?

JVM (Java Virtual Machine):	**JRE (Java Runtime Environment)**:	**JDK (Java Development Kit)**:
Purpose: It provides a runtime environment where Java bytecode can be executed.	**Purpose**: It provides the libraries and other components necessary to run Java applications.	**Purpose**: It provides the complete environment for both developing and running Java programs.
Functionality: It converts Java bytecode into machine-specific code and manages system resources during program execution, such as memory management and garbage collection.	**Functionality**: You can run Java applications on a system that has only the JRE installed, but you cannot compile Java programs.	**Functionality**: You need the JDK to write, compile, and run Java programs.
Platform-Dependent: The JVM is platform-specific, meaning there is a separate JVM for each operating system (Windows, Linux, etc.).	**Components**: Includes the JVM and core libraries (like rt.jar), but it does **not** contain tools for developing Java applications (like compilers).	**Components**: Includes the JRE (which contains the JVM) along with development tools like a compiler (javac), debugger, and other utilities necessary for writing, debugging, and compiling Java programs.
Key Point: JVM is part of both JDK and JRE but cannot execute Java programs on its own without the JRE or JDK.	**Key Point**: The JRE is used to **run** Java programs but not to develop them.	**Key Point**: The JDK is for **developing** Java applications, whereas the JRE is just for running them.

2. Explain the concept of inheritance in Java.

=> Inheritance in Java is one of the fundamental principles of Object-Oriented Programming (OOP). It allows a new class (child or subclass) to inherit properties and behaviors (fields and methods) from an existing class (parent or superclass). This promotes code reusability and establishes a hierarchical relationship between classes.

Key Concepts of Inheritance:

1. **Parent Class (Superclass)**: The class whose properties and methods are inherited by another class.
2. **Child Class (Subclass)**: The class that inherits properties and methods from the parent class.

Types of Inheritance in Java:

1. **Single Inheritance**: A subclass inherits from one parent class.

- Example:

```
class Animal {
        void eat() { System.out.println("Animal is eating"); }
    }
    class Dog extends Animal {
        void bark() { System.out.println("Dog is barking"); }
    }
```

2. **Multilevel Inheritance**: A class is derived from a class, which is already derived from another class (i.e., a chain of inheritance).

   - Example:

```
class Animal {
    void eat() { System.out.println("Animal is eating"); }
}
class Dog extends Animal {
    void bark() { System.out.println("Dog is barking"); }
}
class Puppy extends Dog {
    void weep() { System.out.println("Puppy is weeping"); }
}
```

3. **Hierarchical Inheritance**: Multiple classes inherit from the same parent class.

   - Example:

```
class Animal {
    void eat() { System.out.println("Animal is eating"); }
}
class Dog extends Animal {
    void bark() { System.out.println("Dog is barking"); }
}
class Cat extends Animal {
    void meow() { System.out.println("Cat is meowing"); }
}
```

###Definition wih example:
1.Method Overriding: A subclass can modify or provide a specific implementation for a method that is already defined in the parent class. This is known as **method overriding**.

- Example:

```
class Animal {
    void sound() { System.out.println("Animal makes sound"); }
}
class Dog extends Animal {
    @Override
    void sound() { System.out.println("Dog barks"); }
}
```

2.Super Keyword: The super keyword is used to call the parent class's methods and constructors. It is particularly useful when the subclass overrides methods or has a constructor that needs to refer to the parent class.

- Example:

```
class Animal {
   void eat() { System.out.println("Animal is eating"); }
}
class Dog extends Animal {
   void eat() {
      super.eat();  // Calls the parent class's eat method
      System.out.println("Dog is eating");
   }
}
```

4. What are constructors in Java?

=> Constructors in Java are special methods used to initialize objects. They are called automatically when an object is created using the new keyword. The main purpose of a constructor is to set the initial state of an object by assigning values to its fields.

Key Features of Constructors:

1. **Same Name as Class**: A constructor must have the same name as the class in which it is defined.
2. **No Return Type**: Constructors do not have a return type, not even void.
3. **Automatically Invoked**: They are called automatically when an object is created.
4. **Overloading**: Constructors can be overloaded, meaning a class can have multiple constructors with different parameters.

Types of Constructors:

1. **Default Constructor (No-Arg Constructor)**:
   - If no constructor is explicitly defined in a class, Java provides a default constructor.
   - It initializes the object with default values (e.g., null for objects, 0 for integers, false for booleans).
   - If a constructor with parameters is defined, the default constructor will no longer be provided automatically.

Example:

```
class Person {
    String name;
    int age;

    // Default constructor
    Person() {
        this.name = "Unknown";
        this.age = 0;
    }
}

public class Main {
    public static void main(String[] args) {
        Person person = new Person();  // Default constructor is called
        System.out.println(person.name);  // Output: Unknown
        System.out.println(person.age);   // Output: 0
    }
}
```

2. **Parameterized Constructor**:
    - A parameterized constructor takes arguments and initializes the object with custom values provided during the creation of the object.

Example:

```
class Person {
    String name;
    int age;

    // Parameterized constructor
    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}


public class Main {
    public static void main(String[] args) {
        Person person = new Person("John", 25);  // Parameterized constructor is called
        System.out.println(person.name);  // Output: John
        System.out.println(person.age);   // Output: 25
    }
}
```

###Definition:

1. **Constructor Overloading:**

- A class can have multiple constructors with different parameter lists. This is called **constructor overloading**. The appropriate constructor is called based on the arguments passed when creating an object.

Example:

```
class Person {
    String name;
    int age;

    // Default constructor
    Person() {
        this.name = "Unknown";
        this.age = 0;
    }

    // Parameterized constructor
    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    // Another parameterized constructor
    Person(String name) {
        this.name = name;
        this.age = 18;  // Default age
    }
}

public class Main {
    public static void main(String[] args) {
        Person p1 = new Person();  // Calls default constructor
        Person p2 = new Person("Alice", 30);  // Calls parameterized constructor
        Person p3 = new Person("Bob");  // Calls another parameterized constructor

        System.out.println(p1.name + ", " + p1.age);  // Output: Unknown, 0
        System.out.println(p2.name + ", " + p2.age);  // Output: Alice, 30
        System.out.println(p3.name + ", " + p3.age);  // Output: Bob, 18
    }
}
```

2. **Constructor Chaining:**

- **Constructor chaining** occurs when one constructor calls another constructor of the same class. It is achieved using the this() keyword.

Example:

```
class Person {
    String name;
    int age;

    // Default constructor
    Person() {
        this("Unknown", 0);  // Calls the parameterized constructor
    }


    // Parameterized constructor
    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

3. What are final, finally, and finalize in Java?

⇨ **1. final (Keyword)**

Purpose: The final keyword is used to restrict the modification of classes, methods, and variables.

Usage:

Final Class: A class marked as final cannot be subclassed (i.e., no other class can extend it).

Example:

```
public final class Animal {
    // This class cannot be inherited
}
```

Final Method: A method marked as final cannot be overridden by subclasses.

Example:

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

Final Variable: A variable marked as final cannot be reassigned once initialized. If it is a primitive, its value cannot change, and if it is an object, its reference cannot change, but its internal state can.

Example:

```
public class Example {
    final int age = 25;  // Can't change the value of 'age'
}
```

finally (Block)

- **Purpose**: The finally block is used in exception handling to execute code regardless of whether an exception is thrown or not. It is typically used to release resources (e.g., closing files, releasing database connections).
- **Use**: The finally block follows a try-catch block and is always executed, even if an exception occurs or a return statement is encountered.
    - Example:

```
public class Example {
    public static void main(String[] args) {
        try {
            int result = 10 / 0;  // This will throw an exception
        } catch (ArithmeticException e) {
            System.out.println("Caught an exception");
        } finally {
            System.out.println("Finally block is executed");  // Always executed
        }
    }
}
```

3. finalize (Method)

- **Purpose**: The finalize method is used to perform cleanup operations before an object is garbage collected. It is called by the garbage collector when it determines that there are no more references to the object.
- **Usage**: You can override the finalize() method in your class to release system resources or perform any cleanup tasks before the object is destroyed. However, the use of finalize() is generally discouraged, as it is unpredictable when it will be called and has performance implications. From Java 9 onwards, finalize() has been deprecated.
    - Example:

```
public class Example {
    @Override
    protected void finalize() throws Throwable {
        System.out.println("Finalize method called before garbage collection");
    }
}

public class Main {
    public static void main(String[] args) {
        Example obj = new Example();
        obj = null;  // Dereferencing the object
        System.gc();  // Requesting garbage collection
    }
}
```

4. Explain the difference between method overloading and method overriding.

=>

Points	Method Overloading	Method Overriding
Definition	Having multiple methods with the same name but different parameter lists in the same class.	Providing a specific implementation of a method in a subclass that already exists in the parent class.
Where it occurs	Within the same class.	Between a superclass and a subclass.
Parameters	Must be different (in number, type, or order).	Must be exactly the same.
Return Type	Can be different but usually the same.	Must be the same.
Binding	Compile-time (Static Binding).	Run-time (Dynamic Binding).
Purpose	To allow methods to handle different data types or numbers of arguments.	To modify or extend the behavior of a parent class's method.
Inheritance	Not required.	Requires inheritance.
Annotation	Not needed.	@Override annotation is recommended.
Polymorphism	Example of compile-time polymorphism.	Example of run-time polymorphism.

5. What is a static method in Java?

=> A static method in Java belongs to the class rather than any particular instance of the class. It can be called directly using the class name, without needing to create an instance. Static methods are typically used for operations that do not require any data from instances (objects) of the class. They can only access other static data and static methods directly.

7. How does garbage collection work in Java?

=> Garbage collection in Java is an automatic process that removes unused or unreferenced objects from memory, freeing up resources for other programs. The Java Virtual Machine (JVM) periodically performs garbage collection to identify objects that are no longer reachable in the application. When it detects an object that no longer has any references pointing to it, it marks it as eligible for garbage collection, and eventually, the memory is reclaimed. This helps in efficient memory management and prevents memory leaks.

8. What are the different access modifiers in Java?

Java has four primary access modifiers:

1. **Public**: The field or method is accessible from any other class.
2. **Protected**: The field or method is accessible within its own package and by subclasses.
3. **Default (Package-private)**: If no modifier is specified, the field or method is accessible only within its own package.
4. **Private**: The field or method is accessible only within its own class.

9. What is a string pool in Java?

=>The string pool, also known as the interned string pool, is a special memory area in Java where string literals are stored. When a new string literal is created, Java checks if an identical string already exists in the pool. If it does, Java reuses the existing string reference rather than creating a new one, saving memory and improving performance. Strings created with the new keyword are not added to the pool by default, although they can be added explicitly with the .intern() method.

10. Explain the difference between == and .equals() in Java.

=> In Java, == is used to compare references, meaning it checks if two object references point to the same memory location. On the other hand, .equals() is a method that is generally overridden to compare the actual contents or values of two objects. For example, with strings, == checks if two string references are the same, while .equals() checks if the contents of the strings are the same.