SADIQ TECH SOLUTIONS

JAVA
ZERO TO HERO

From Novice to Java Virtuoso - Unleash
Your Inner Hero!

M SADIQVA L I

Copyrights

Disclaimer:

Table of Contents

Java Polymorphism

- Method Overloading
- Method Overriding
- Covariant Return Type
- super keyword
- Instance Initializer block
- final keyword
- Runtime Polymorphism
- Dynamic Binding
- instanceof operator

Java Abstraction

- Abstract class
- Interface
- Abstract vs Interface

Java Encapsulation

- Package
- Access ModifiersEncapsulation

Java Array

- Java Array

Java OOPs Misc

- Object class
- Object Cloning
- Math class
- Wrapper Class
- Java Recursion
- Call By Value
- strictfp keyword
- javadoc tool
- Command Line Arg
- Object vs Class
- Overloading vs Overriding

Java String

- What is String
- Immutable String
- String Comparison
- String Concatenation
- Substring
- Methods of String class
- StringBuffer class
- StringBuilder class
- String vs StringBuffer
- StringBuffer vs Builder
- Creating Immutable classtoString methodStringTokenizer class

Java Regex

- Matcher class
- Pattern class
- Example of Java Regular Expressions
- Java regex API

Exception Handling

- Java Exceptions
- Java Try-catch block
- Java Multiple Catch Block
- Java Nested try
- Java Finally Block
- Java Throw Keyword
- Java Exception Propagation
- Java Throws Keyword
- Java Throw vs Throws
- Final vs Finally vs Finalize
- Exception Handling with Method Overriding
- Java Custom Exceptions

Java Inner Class

- What is inner class
- Member Inner class
- Anonymous Inner class
- Local Inner class
- static nested class
- Nested Interface

Java Multithreading

- What is Multithreading
- Life Cycle of a Thread
- How to Create Thread
- Thread Scheduler
- Sleeping a thread
- Start a thread twice
- Calling run() method
- Joining a thread
- Naming a thread
- Thread Priority
- Daemon Thread
- Thread Pool
- Thread Group
- ShutdownHook
- Performing multiple task
- Garbage Collection
- Runtime class

Java Synchronisation

- Synchronisation in java
- Synchronised block
- static synchronisation
- Deadlock in Java
- Inter-thread Comm
- Interrupting Thread
- Reentrant Monitor

Java I/O

- Java Input/Output
- FileOutputStream
- FileInputStream
- BufferedOutputStream
- BufferedInputStream
- SequenceInputStream
- ByteArrayOutputStream
- ByteArrayInputStream
- DataOutputStream
- DataInputStream
- Java FilterOutputStream
- Java FilterInputStream
- Java ObjectStream
- Java ObjectStreamField
- Console
- FilePermission

- Java HashSet
- Java LinkedHashSet
- Java TreeSet
- Queue & PriorityQueue
- Deque & ArrayDeque
- Java Map Interface
- Java HashMap
- Working of HashMap
- Java LinkedHashMap
- Java TreeMap
- Java Hashtable
- HashMap vs Hashtable
- Java EnumSet
- Java EnumMap
- Collections class
- Sorting Collections
- Comparable interface
- Comparator interface
- Comparable vs Comparator
- Properties class
- ArrayList vs Vector
- Java Vector
- Java Stack
- Java Collection Interface
- Java Iterator Interface
- Java Deque Interface
- Java ConcurrentHashMap
- Java ConcurrentLinkedQueue

Java Zero to Hero

Java Basics

What is Java?

Java is a high-level, object-oriented programming language that is designed to be portable, secure, and robust. It was developed by James Gosling and his team at Sun Microsystems (later acquired by Oracle Corporation) and first released in 1995. Java is known for its "write once, run anywhere" (WORA) principle, which means that Java code can be written once and executed on any platform that has a Java Virtual Machine (JVM) installed, without the need for recompilation. This platform independence is achieved by using the Java bytecode format, which is an intermediate representation of the code that is platform-neutral.

Java is widely used for building various types of applications, including web applications, desktop applications, mobile apps, enterprise systems, and more. It has become one of the most popular programming languages due to its versatility, ease of use, and robustness.

History of Java

- **1991**: The development of Java began under the name "Oak" at Sun Microsystems by James Gosling and his team. The initial focus was on creating a programming language for consumer electronic devices.
- **1995**: The first public release of Java (JDK 1.0) was made on May 23, 1995. It included the core features of the Java language and the Java applet technology, which allowed small Java programs to be embedded into web pages.
- **1996**: JDK 1.1 was released, introducing new features and improvements to the language and the standard library.
- **1998**: JDK 1.2 (Java 2) was released, bringing significant updates to the language and the platform, including the introduction of the Swing GUI toolkit and the Collections Framework.
- **2004**: JDK 1.5 (Java 5) was released, introducing several important language features like generics, enhanced for loop, autoboxing, and annotations.
- **2011**: Oracle Corporation acquired Sun Microsystems, becoming the new steward of the Java platform.

- **2014**: JDK 1.8 (Java 8) was released, introducing lambda expressions, the Stream API, and other language enhancements.
- **2017**: JDK 9 introduced modularity with the Java Platform Module System (JPMS).
- **2020**: JDK 14, 15, 16, and 17 were released successively, each bringing new features and improvements to the language and platform.

Features of Java

1. **Platform Independence**: Java code is compiled into bytecode, which is executed by the JVM. This bytecode can run on any platform with a compatible JVM, making Java platform-independent.
2. **Object-Oriented**: Java follows the object-oriented programming paradigm, supporting concepts like classes, objects, inheritance, polymorphism, and encapsulation.
3. **Garbage Collection**: Java has an automatic garbage collection mechanism that manages memory, freeing developers from manual memory management.
4. **Robustness**: Java's strong type-checking, exception handling, and memory management contribute to its robustness.
5. **Security**: Java provides a secure execution environment by using a sandbox model for applets and built-in security mechanisms.
6. **Multi-threading Support**: Java has built-in support for concurrent programming through threads, enabling the creation of multithreaded applications.
7. **Rich Standard Library**: Java comes with a vast standard library that provides a wide range of functionalities, making development faster and more efficient.
8. **High Performance**: Although Java is an interpreted language, its bytecode is compiled at runtime by the JVM, resulting in high performance.

C++ vs. Java

C++ and Java are both widely used programming languages, but they have some fundamental differences:

- **Syntax**: C++ uses a more complex syntax compared to Java. Java was designed to have a simpler and more user-friendly syntax.
- **Memory Management**: C++ requires manual memory management using pointers, whereas Java has automatic garbage collection, making it less prone to

memory-related bugs.

- **Platform Independence**: C++ code is compiled into platform-specific machine code, while Java code is compiled into platform-independent bytecode.
- **Performance**: C++ is generally considered to have better performance than Java because it is compiled directly to machine code. However, Java performance has improved significantly with advancements in JIT (Just-In-Time) compilation.
- **Object-Oriented Features**: Both languages support object-oriented programming, but Java enforces stricter object-oriented principles, like not allowing multiple inheritances and using interfaces.
- **Standard Library**: Java has a more extensive and consistent standard library compared to C++, which can sometimes have a more fragmented ecosystem due to various libraries and implementations.

Hello Java Program

Below is a simple "Hello World" program in Java:

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

Explanation:

- We define a class named `HelloWorld`. In Java, all code must be inside classes.
- The `public` keyword indicates that the class is accessible from other classes.
- The `static` keyword is used to define a static method that belongs to the class itself, not to any specific instance of the class.
- The `void` keyword indicates that the `main` method does not return any value.
- `main` is the entry point of the program. It is the method that will be executed when we run the program.
- The `String[] args` is an array of strings representing the command-line arguments passed to the program.
- `System.out.println()` is a method used to print the text "Hello, Java!" to the console.

Program Internal

The "Program Internal" is not a standard term in Java. However, if you are referring to the internal structure of a Java program, we can briefly discuss the Java compilation and execution process.

1. **Java Compilation Process**:
   - Java source code is written in `.java` files.
   - The Java compiler (`javac`) translates the human-readable Java source code into platform-independent bytecode. The bytecode is stored in `.class` files.
2. **Java Execution Process**:
   - The Java Virtual Machine (JVM) is responsible for executing Java bytecode.
   - When you run a Java program, the JVM loads the necessary classes and starts executing the `main` method of the specified class (the entry point of the program).
3. **Java Bytecode**:
   - Java bytecode is an intermediate representation of the Java source code. It is not directly executed by the operating system but by the JVM.
   - The bytecode consists of instructions that the JVM can understand and execute.

It is important to note that understanding the internal details of the Java program, such as bytecode, is not usually required for typical Java programming tasks. However, having a basic understanding of these concepts can be helpful when dealing with advanced topics or troubleshooting performance issues.

Setting Path:

Setting the path in Java is essential to make the Java Development Kit (JDK) tools, including the Java compiler (`javac`) and Java runtime (`java`), accessible from any directory on your computer. The path allows the operating system to find these tools when you run Java commands from the command prompt or terminal.

Here's how to set the path in Windows and Unix-based systems (Linux and macOS):

Windows:

1. Find the JDK installation directory. It usually looks like `C:\\Program Files\\Java\\jdk_version`.
2. Right-click on "This PC" or "My Computer" and select "Properties."

3. Click on "Advanced system settings" (on the left side of the window).
4. In the System Properties window, click the "Environment Variables" button.
5. Under "System Variables," find the "Path" variable, and click "Edit."
6. Click "New" and add the path to the JDK's "bin" directory (e.g., `C:\\Program Files\\Java\\jdk_version\\bin`).
7. Click "OK" on all the windows to save the changes.

Unix-based Systems (Linux and macOS):

1. Open the terminal.
2. Find the JDK installation directory. It's usually located at `/usr/lib/jvm/jdk_version`.
3. Edit the `.bashrc` or `.bash_profile` file in your home directory using a text editor (e.g., `nano`, `vi`, or `gedit`).
4. Add the following line to the file:
   Replace `/usr/lib/jvm/jdk_version` with the actual path to your JDK installation.

   ```
   export PATH="/usr/lib/jvm/jdk_version/bin:$PATH"
   ```

5. Save the file and exit the text editor.
6. Run `source ~/.bashrc` or `source ~/.bash_profile` in the terminal to apply the changes to the current session.

After setting the path, you can verify that it worked by opening a new terminal or command prompt and running `java -version` and `javac -version` commands. They should display the version information of your installed JDK.

JDK, JRE, and JVM:

- **JDK (Java Development Kit)**: JDK is a software development kit provided by Oracle (or other vendors) that includes tools needed to develop, compile, and run Java applications. It contains the Java compiler (`javac`), Java runtime (`java`), Java documentation generator (`javadoc`), and other utilities. JDK is essential for Java developers.
- **JRE (Java Runtime Environment)**: JRE is a subset of JDK and is required to run Java applications. It includes the Java Virtual Machine (JVM) and the Java class libraries needed for executing Java bytecode. Users who only need to run Java applications but not develop them can install the JRE.

- **JVM (Java Virtual Machine)**: JVM is the heart of the Java platform. It is responsible for executing Java bytecode. JVM is an abstract machine that provides a runtime environment in which Java bytecode can be executed. JVM implementations are platform-specific, allowing Java programs to be platform-independent.

Java Variables:

In Java, variables are used to store data values that can be manipulated within a program. Before using a variable, it needs to be declared with a specific data type. Java supports the following variable types:

1. **Primitive Data Types**: These are basic data types built into the language.
    - Numeric Types: `byte` , `short` , `int` , `long` , `float` , and `double` .
    - Characters: `char` .
    - Boolean: `boolean` .
2. **Reference Data Types**: These types are references to objects in memory.
    - Examples: `String` , user-defined classes, arrays, etc.

Java Data Types:

1. **Primitive Data Types**:
    - `byte` : 8-bit signed integer. Range: -128 to 127.
    - `short` : 16-bit signed integer. Range: -32,768 to 32,767.
    - `int` : 32-bit signed integer. Range: -2^31 to 2^31 - 1.
    - `long` : 64-bit signed integer. Range: -2^63 to 2^63 - 1.
    - `float` : 32-bit floating-point. Can represent decimal numbers.
    - `double` : 64-bit floating-point. More precise than `float` .
    - `char` : 16-bit Unicode character. Represents single characters.
    - `boolean` : Represents `true` or `false` .
2. **Reference Data Types**:
    - `String` : A sequence of characters.
    - Arrays: Collections of elements of the same type.

Unicode System:

Unicode is a character encoding standard that aims to support all the characters from all the writing systems in the world. In Java, characters are represented using the Unicode character set, allowing Java to be used with multiple languages and character sets.

For example, you can declare a Unicode character in Java like this:

```
char unicodeChar = '\\u0041'; // Represents the character 'A'
```

The escape sequence `\\u` is used to indicate a Unicode character, followed by the four-digit hexadecimal representation of the Unicode code point.

Operators:

Java supports various types of operators, which are used to perform operations on variables and values. Here are some common operators in Java:

- **Arithmetic Operators**: `+` (addition), (subtraction), (multiplication), `/` (division), `%` (modulus).
- **Assignment Operators**: `=`, `+=`, `=`, `=`, `/=`, `%=`.
- **Increment and Decrement Operators**: `++` (increment), `-` (decrement).
- **Relational Operators**: `==` (equals), `!=` (not equals), `>`, `<`, `>=`, `<=`.
- **Logical Operators**: `&&` (logical AND), `|` (logical OR), `!` (logical NOT).
- **Bitwise Operators**: `&` (bitwise AND), `|` (bitwise OR), `^` (bitwise XOR), `~` (bitwise NOT).

Keywords:

Keywords in Java are reserved words that have predefined meanings and cannot be used as identifiers (e.g., variable names, class names). They are an integral part of the language and serve specific purposes. Some examples of Java keywords include:

```
abstract, boolean, break, byte, case, catch, class, continue, default, do, double,
else, extends, final, finally, float, for, if, implements, import, instanceof,
int, interface, long, new, package, private, protected, public, return, short,
static, super, switch, this, throw, throws, try, void, while
```

These keywords are used for various purposes, such as defining classes, control flow, variable declaration, method definition, and more. It's essential to be familiar with these keywords when programming in Java.

Java Control Statements:

Control statements in Java are used to control the flow of a program's execution based on certain conditions or iterations. They help make decisions and repeat actions as needed.

Java If-else:

The `if-else` statement is used to execute a block of code conditionally. If the given condition is true, the code inside the `if` block is executed; otherwise, the code inside the `else` block (if provided) is executed.

```
int num = 10;

if (num > 0) {
    System.out.println("The number is positive.");
} else {
    System.out.println("The number is non-positive.");
}
```

Java Switch:

The `switch` statement is used to perform different actions based on the value of a variable or an expression. It provides an alternative to using multiple `if-else` statements.

```java
int dayOfWeek = 3;

switch (dayOfWeek) {
    case 1:
        System.out.println("Monday");
        break;
    case 2:
        System.out.println("Tuesday");
        break;
    case 3:
        System.out.println("Wednesday");
        break;
    default:
        System.out.println("Invalid day");
}
```

Java For Loop:
The `for` loop is used to execute a block of code repeatedly for a fixed number of times. It consists of three parts: initialization, condition, and increment or decrement.

```java
for (int i = 1; i <= 5; i++) {
    System.out.println("Iteration: " + i);
}
```

Java While Loop:

The `while` loop repeatedly executes a block of code as long as the given condition is true.

```java
int count = 1;

while (count <= 5) {
    System.out.println("Count: " + count);
    count++;
}
```

/*

Java Do-While Loop:

The **Do-while** loop is similar to the **while** loop, but ensures that the block of code is executed at least once before checking the condition.

```
int x = 1;

do {
    System.out.println("Value of x: " + x);
    x++;
} while (x <= 5);
```

Java Break:

The `break` statement is used to terminate the loop or switch statement prematurely, based on a condition.

```
for (int i = 1; i <= 10; i++) {
    if (i == 5) {
        break; // Terminate the loop when i equals 5
    }
    System.out.println("i: " + i);
}
```

Java Continue:

The `continue` statement is used to skip the current iteration and continue with the next iteration of a loop based on a condition.

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) {
        continue; // Skip the iteration when i equals 3
    }
    System.out.println("i: " + i);
}
```

Java Comments:

Comments in Java are used to provide explanations or documentation within the code. They are ignored by the compiler and not executed as part of the program.

- Single-line comments start with `//` and continue until the end of the line:

```
// This is a single-line comment.
```

- Multiline comments start with **/*** and end with */ , it allows to multiple lines to comment.

```
/*
This is a multi-line comment.
It spans across multiple lines.
*/
```

- Javadoc comments are used for generating documentation. They start with `/**` and end with `/`. Javadoc comments are placed before classes, methods, or fields and can include additional tags for documentation generation:

```
/**
 * This method calculates the sum of two numbers.
 * @param a The first number.
 * @param b The second number.
 * @return The sum of a and b.
 */
public int sum(int a, int b) {
    return a + b;
}
```

Using comments effectively helps improve code readability and makes it easier for other developers to understand your code.

Java Object Class:

In Java, the `Object` class is the root class of all other classes. It is defined in the `java.lang` package and is automatically inherited by all other classes, either directly or indirectly. Every class in Java is a subclass of the `Object` class.

The `Object` class provides several important methods that are inherited by all classes, such as `equals()`, `hashCode()`, `toString()`, `getClass()`, and more. These methods can be overridden in user-defined classes to provide custom behavior.

```
public class MyClass {
    // MyClass inherits from the Object class implicitly
}
```

Java OOPs Concepts:

Java is an object-oriented programming language, and its primary focus is on the concepts of Object-Oriented Programming (OOP). Some of the key OOPs concepts in Java include:

1. **Encapsulation**: Encapsulation is the bundling of data (attributes) and methods that operate on that data within a single unit, called a class. It helps in data hiding and prevents direct access to the internal data from outside the class.
2. **Inheritance**: Inheritance allows a class (subclass) to inherit the properties and behaviors of another class (superclass). It promotes code reusability and allows new classes to extend existing classes.
3. **Polymorphism**: Polymorphism allows an object to take on multiple forms. It can be achieved through method overloading (compile-time polymorphism) and method overriding (run-time polymorphism).
4. **Abstraction**: Abstraction involves the creation of abstract classes and interfaces, which define the structure and behavior of classes without specifying their implementation details.
5. **Association**: Association represents a relationship between two classes, where one class uses the functionalities provided by another class.
6. **Aggregation**: Aggregation is a special form of association that represents a "has-a" relationship. It implies a relationship where one class contains another class as part of its structure.

7. **Composition**: Composition is a stronger form of aggregation, where one class is composed of one or more other classes. The composed classes cannot exist independently.

Naming Convention:

In Java, following a consistent and standard naming convention is crucial for code readability and maintainability. Here are some common Java naming conventions:

1. **Class Names**: Start with an uppercase letter, and use camel case (capitalize the first letter of each word) for multiple words. Example: `MyClass`, `CarModel`.
2. **Variable Names**: Start with a lowercase letter, and use camel case for multiple words. Example: `age`, `firstName`.
3. **Constant Names**: Use uppercase letters and separate words with underscores. Example: `MAX_VALUE`, `PI`.
4. **Method Names**: Start with a lowercase letter, and use camel case for multiple words. Example: `calculateArea`, `getName`.

Object and Class:

In Java, a class is a blueprint or a template for creating objects. It defines the attributes (fields) and behaviors (methods) that the objects of the class will have. An object is an instance of a class, representing a specific entity based on the class definition.

For example, consider a `Car` class:

```
public class Car {
    String brand;
    String model;

    void startEngine() {
        // Code to start the car's engine
    }
}
```

Here, `Car` is a class, and `brand`, `model`, and `startEngine()` are its members. When you create an object of the `Car` class, it represents a specific car with its own brand and model.

Method:

A method in Java is a block of code that defines the behavior of an object. It is defined within a class and can be invoked on objects of that class. Methods are used to perform specific actions or operations.

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

In this example, `add()` is a method of the `Calculator` class that takes two integer parameters and returns their sum.

Constructor:

A constructor in Java is a special method used to initialize objects of a class. It has the same name as the class and is called automatically when an object is created. Constructors are used to set initial values for the object's attributes or perform other setup tasks.

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

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

In this example, the `Person` class has a constructor that takes the `name` and `age` as parameters and initializes the corresponding attributes.

static Keyword:

The `static` keyword is used to declare members (variables and methods) that belong to the class itself, rather than to individual objects. They are shared by all objects of the class and can be accessed directly using the class name, without creating an instance of the class.

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

In this example, the `add()` method is declared as `static`, so you can call it using `MathUtils.add(2, 3)` without creating an instance of the `MathUtils` class.

this Keyword:

The `this` keyword in Java is a reference to the current instance of the class. It is used to distinguish between instance variables and parameters with the same name, and to access instance variables and methods within the class.

```
public class Person {
    String name;

    public void setName(String name) {
        this.name = name;
    }
}
```

In this example, `this.name` refers to the instance variable `name`, while `name` refers to the method parameter `name`.

Java Inheritance:

Inheritance is one of the fundamental concepts of object-oriented programming (OOP) that allows one class (subclass or derived class) to inherit the properties and behaviors of another class (superclass or base class). The subclass can extend the functionality of the superclass by adding new attributes and methods or by overriding existing ones.

In Java, inheritance is achieved using the `extends` keyword.

IS-A Relationship (Inheritance):

Inheritance represents an IS-A relationship between classes. If Class B inherits from Class A, it means that objects of Class B are also objects of Class A, and they share a common "is-a" relationship.

Consider the following example:

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

class Dog extends Animal {
    void makeSound() {
        System.out.println("Dog barks.");
    }
}
```

In this example, `Dog` is a subclass of `Animal`. So, a `Dog` IS-A `Animal`. When we create a `Dog` object, it can call methods defined in both `Dog` and `Animal`.

```
Animal animal = new Dog();
animal.makeSound(); // Output: Dog barks.
```

Java Aggregation:

Aggregation represents a HAS-A relationship between classes, where one class contains another class as part of its structure. It is a form of association that implies a relationship between a whole and its parts.

In Java, aggregation is achieved by creating a class with a reference to another class.

Consider the following example:

```
class Address {
    String street;
    String city;

    Address(String street, String city) {
        this.street = street;
        this.city = city;
    }
}

class Person {
    String name;
    Address address;

    Person(String name, Address address) {
        this.name = name;
        this.address = address;
    }
}
```

In this example, `Person` has a HAS-A relationship with `Address`, as `Person` contains an `Address` object as one of its attributes.

```
Address address = new Address("Main Street", "New York");
Person person = new Person("John Doe", address);
```

Here, the `person` object contains an `address` object, forming an aggregation relationship.

In summary, inheritance (IS-A) is used to achieve code reusability and represents a relationship between a superclass and its subclasses, while aggregation (HAS-A) represents a relationship between a whole class and its parts, allowing one class to contain another as an attribute. Both concepts are essential in designing object-oriented systems and promoting modularity and flexibility in code.

Java Polymorphism:

Polymorphism is a key concept in object-oriented programming (OOP) that allows objects of different classes to be treated as objects of a common superclass. It provides a way to perform a single action in different ways based on the context. There are two main types of polymorphism in Java: compile-time polymorphism and runtime polymorphism.

Method Overloading:

Method overloading is an example of compile-time (static) polymorphism. It allows a class to have multiple methods with the same name but different parameters. The compiler determines which method to call based on the number or types of arguments passed.

```
class MathUtils {
    int add(int a, int Constructor {
        return a + b;
    }

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

In this example, the `add()` method is overloaded with two different parameter types (integers and doubles).

Method Overriding:

Method overriding is an example of runtime (dynamic) polymorphism. It occurs when a subclass provides a specific implementation for a method that is already defined in its superclass. The method in the subclass must have the same name, return type, and parameters as the method in the superclass.

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

class Dog extends Animal {
    void makeSound() {
        System.out.println("Dog barks.");
    }
}
```

In this example, the `Dog` class overrides the `makeSound()` method of the `Animal` class to provide a specific implementation for dogs.

Covariant Return Type:

Covariant return type is a feature introduced in Java 5 that allows a subclass to override a method in its superclass with a more specific return type. The return type in the subclass method can be a subclass of the return type in the superclass method.

```
class Animal {
    Animal getAnimal() {
        return this;
    }
}

class Dog extends Animal {
    Dog getAnimal() {
        return this;
    }
}
```

In this example, the `getAnimal()` method in the `Dog` class returns a `Dog` object, which is a subclass of the `Animal` class's return type.

super Keyword:

The `super` keyword is used to refer to the superclass's members (fields, methods, and constructors) from the subclass. It is often used to call the superclass's constructor or to access overridden methods or fields.

```
class Animal {
    String name;

    Animal(String name) {
        this.name = name;
    }
}

class Dog extends Animal {
    String breed;

    Dog(String name, String breed) {
        super(name); // Call the superclass constructor
        this.breed = breed;
    }
}
```

In this example, the `super(name)` call in the `Dog` constructor is used to invoke the `Animal` class's constructor with the `name` parameter.

Instance Initializer Block:

The instance initializer block is a block of code that is executed whenever an instance of the class is created, before the constructor is called. It is defined within curly braces without any method name.

```
class MyClass {
    {
        // Instance initializer block
        System.out.println("Instance initializer block is executed.");
    }

    MyClass() {
        System.out.println("Constructor is called.");
    }
}
```

When you create an object of the `MyClass`, the instance initializer block is executed before the constructor.

final Keyword:

The `final` keyword is used to declare that a variable, method, or class cannot be changed or overridden after its initialization or definition.

- Final Variable: Once initialized, the value of a final variable cannot be modified.

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

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

```
name; Parent {
    final void display() {
        // Method implementation
    }
}
```

- Final Class: A final class cannot be subclassed.

```
final class MyClass {
    String model; Class definition
}
```

Runtime Polymorphism:

Runtime polymorphism is also known as dynamic polymorphism or late binding. It occurs when the method to be executed is determined at runtime based on the actual type of the object, not at compile time.

In Java, method overriding is an example of runtime polymorphism, where the decision about which method to call is made during runtime.

Dynamic Binding:

Dynamic binding is the process of linking a method call to its method definition during runtime. It allows Java to support runtime polymorphism and method overriding.

When a method is overridden in a subclass, the actual method to be called is determined dynamically based on the type of the object at runtime.

instanceof Operator:

The `instanceof` operator is used to check whether an object belongs to a specific class or its subclasses. It returns `true` if the object is an instance of the specified class or `false` otherwise.

```
class Animal { }

class Dog extends Animal { }

public class Main {
    public static void main(String[] args) {
        Animal animal = new Dog();

        if (animal instanceof Dog) {
            System.out.println("The animal is a Dog.");
        }
    }
}
```

In this example, `animal instanceof Dog` checks whether the `animal` object is an instance of the `Dog` class or any of its subclasses. Since `animal` is an instance of `Dog`, the output will be "The animal is a Dog."

Java Abstraction:

Abstraction is one of the core principles of object-oriented programming (OOP) and refers to the process of hiding the implementation details of an object and exposing only the relevant features or behaviors to the outside world. It allows us to focus on what an object does rather than how it does it. Abstraction is achieved in Java through abstract classes and interfaces.

Abstract Class:

An abstract class is a class that cannot be instantiated directly and is intended to be subclassed. It serves as a blueprint for other classes and can contain abstract methods (methods without a body) that must be implemented by its subclasses. An abstract class can also have concrete methods with an implementation.

```
abstract class Shape {
    int x, y;

    abstract void draw();

    void moveTo(int newX, int newY) {
        this.x = newX;
        this.y = newY;
    }
}
```

In this example, `Shape` is an abstract class that has an abstract method `draw()` and a concrete method `moveTo()`. Classes that inherit from `Shape` must provide an implementation for the `draw()` method.

Interface:

An interface in Java is a blueprint of a class that defines a contract for the classes that implement it. It contains only abstract methods and constants (implicitly `public`, `static`, and `final`). Interfaces are used to achieve multiple inheritance in Java.

Note:

 This is a **preview Java Core & Advanced e-Book** containing **only 30 pages**.
 It is provided to help you understand **how the Java Core & Advanced e-Book and is structured**.
 The **complete Java Full Core & Advanced** includes detailed concepts, real-world examples, and career guidance.
 Buy now from our official website: https://topitcourses.com/java-core-advanced/