Java final keyword

In Java, the final keyword is used to denote constants. It can be used with variables, methods, and classes. Once any entity (variable, method or class) is declared final, it can be assigned only once. That is,

- the final variable cannot be reinitialized with another value
- the final method cannot be overridden
- the final class cannot be extended

1. Java final Variable

In Java, we cannot change the value of a final variable.

Example:

```
class Main {
  public static void main(String[] args) {
    // create a final variable
    final int AGE = 32;
    // try to change the final variable
    AGE = 45;
    System.out.println("Age: " + AGE);
  }
}
```

When we run the program, we will get a compilation error with the following message.

```
cannot assign a value to final variable AGE
    AGE = 45;
    ^
```

Note: It is recommended to use uppercase to declare final variables in Java.

2. Java final Method

In Java, the final method cannot be overridden by the child class.

Example:

```
class FinalDemo {
    // create a final method
    public final void display() {
      System.out.println("This is a final method.");
    }
}

class Main extends FinalDemo {
  // try to override final method
  public final void display() {
    System.out.println("The final method is overridden.");
  }

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

In the above example, we have created a final method named display() inside the FinalDemo class. Here, the Main class inherits the FinalDemo class.

We have tried to override the final method in the Main class. When we run the program, we will get a compilation error with the following message.

```
 display() in Main cannot override display() in FinalDemo
  public final void display() {
                    ^
  overridden method is final
```

3. Java final Class

In Java, the final class cannot be inherited by another class.

Example:

```
// create a final class
final class FinalClass {
  public void display() {
    System.out.println("This is a final method.");
  }
}

// try to extend the final class
class Main extends FinalClass {
  public  void display() {
    System.out.println("The final method is overridden.");
  }

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

In the above example, we have created a final class named FinalClass. Here, we have tried to inherit the final class by the Main class.

When we run the program, we will get a compilation error with the following message.

```
cannot inherit from final FinalClass
class Main extends FinalClass {
                   ^
```

Polymorphism in Java

- The derivation of the word Polymorphism is from two different Greek words- poly and morphs. “Poly” means numerous, and “Morphs” means forms. So, polymorphism means innumerable forms.
- **Polymorphism** in Java means "many forms." It allows the same method or operator to behave differently based on the context.
- **Polymorphism** means that the same action can look different depending on the object or situation.
- Polymorphism occurs when there is inheritance, i.e., many classes are related.
- For example, an individual can have different relationships with different people. A woman can be a mother, a daughter, a sister, and a friend, all at the same time, i.e. she performs other behaviors in different situations.

Use cases of Polymorphism:

A. Method Overriding (Run-time Polymorphism)

Runtime polymorphism in Java is also popularly known as Dynamic Binding or Dynamic Method Dispatch. In this process, the call to an overridden method is resolved dynamically at runtime rather than at compile-time. You can achieve Runtime polymorphism via Method Overriding.

Note: Runtime polymorphism can only be achieved through functions and not data members.

Method Overriding happens when a subclass provides a specific implementation of a method that is already defined in its superclass.

Example:

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

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

public class Main {
  public static void main(String[] args) {
    Animal myAnimal = new Dog();
    myAnimal.makeSound(); // Output: Dog barks
  }       }
```

B. Method Overloading (Compile-time Polymorphism)

Compile Time Polymorphism in Java is also known as Static Polymorphism. Furthermore, the call to the method is resolved at compile-time. Compile-Time polymorphism is achieved through Method Overloading. This type of polymorphism can also be achieved through Operator Overloading. However, Java does not support Operator Overloading.

Method overloading is the process that can create multiple methods of the same name in the same class, and all the methods work in different ways. Method overloading occurs when there is more than one method of the same name in the class.

Example:

```
class Printer {
  public void print(int num) {
    System.out.println("Printing number: " + num);
  }

  public void print(String text) {
    System.out.println("Printing text: " + text);
  }
}

public class Main {
  public static void main(String[] args) {
    Printer myPrinter = new Printer();
    myPrinter.print(5); // Output: Printing number: 5
    myPrinter.print("Hello"); // Output: Printing text: Hello
  }
}
```

C. Operator Overloading

Operator Overloading allows operators to perform different functions based on the types of their operands. In Java, this happens automatically for some operators.

Example:

```
int sum = 5 + 10; // Adds two numbers

String greeting = "Hello" + " World"; // Concatenates two strings
```

The + operator adds numbers and concatenates strings based on what it's used with.

D. Coercion

Coercion deals with implicitly converting one type of object into a new object of a different kind. Also, this is done automatically to prevent type errors in the code.

Programming languages such as C, Java, etc support the conversion of value from one data type to another data type. Data type conversions are of two types, i.e., implicit and explicit.

Implicit type conversion is automatically done in the program and this type of conversion is also termed coercion.

For example, if an operand is an integer and another one is in float, the compiler implicitly converts the integer into float value to avoid type error.

Example:

```
class coercion {
        public static void main(String[] args) {
        Double area = 3.14*5*7;
        System.out.println(area);
        String s = "happy";
        int x=5;
        String word = s+x;
        System.out.println(word);
        }
}
```

E. Polymorphic Variables

In Java, the object or instance variables represent the polymorphic variables. This is because any object variables of a class can have an IS-A relationship with their own classes and subclasses.

The Polymorphic Variable is a variable that can hold values of different types during the time of execution.

Parametric polymorphism specifies that while class declaration, a field name can associate with different types, and a method name can associate with different parameters and return types.

Example:

```
class Shape
{
   public void display(){
      System.out.println("A Shape.");
   }
}
class Triangle extends Shape{
   public void display(){
      System.out.println("I am a triangle.");
   }
}
class Main{
   public static void main(String[] args){
      Shape obj;
      obj = new Shape();
      obj.display();
      obj = new Triangle();
      obj.display();
   }
}
```