Day-7: Generics

- What is Generics?
- Difference between Generics and Object type.
- Implementing Generics.
- Generic Bounded Types
- Generic Methods
- Generic Constructor
- Generic Interface
- Wildcard
- Wildcard with upper bound
- Generic restrictions

Generics

- Introduced by JDK 5, generics changed Java in two important ways.
    - It added a new syntactical element to the language.
    - It caused changes to many of the classes and methods in the core API.
- With generics, you can define an algorithm once, independently of any specific type of data, and then apply that algorithm to a wide variety of data types.

What is Generics?

- Generics mean parameterized types.
- Parameterized types enable you to create classes, interfaces, and methods in which the type of data upon which they operate is specified as a parameter.
- A class, interface, or method that operates on a parameterized type is called generic class, generic interface or generic method.

Difference between Generics and Object type

- With Generics, type casting is implicit. But, with Object type type casting is required.

NOTE: Generics works only with Object type, primitive data types cannot be used.

Generic Class Syntax:

```
class class-name<type-param-list > { // …
```

Declaring a reference to a generic class and instance creation:

```
class-name<type-arg-list > var-name = new class-name<type-arg-list >(cons-arg-list);
```

Example:

```
class Gen1<T> {
	T a;
	public Gen1() {
	}
	public Gen1(T a) {
		this.a = a;
	}
	public void setA(T a) {
		this.a = a;
	}
	public T getA() {
        return a;
    }
    public void showType() {
        System.out.println("Type of T is : " + a.getClass().getName());
    }
    public static void main(String args[]) {
        Gen1<Integer> obj = new Gen1<Integer>();
        obj.setA(10);
        System.out.println("Value : " + obj.getA());
        obj.showType();

        Gen1<String> obj1 = new Gen1<String>();
        obj1.setA("Welcome");
        System.out.println("Value : " + obj1.getA());
        obj1.showType();

        // Gen1<int> obj2 = new Gen1<int>();
    }
}
```

Example:

```
class Gen2<T, V> {
    T a;
    V b;
    public Gen2() {
    }
    public Gen2(T a, V b) {
        this.a = a;
        this.b = b;
    }
    public void setA(T a) {
        this.a = a;
    }
    public T getA() {
        return a;
    }
    public void setB(V b) {
        this.b = b;
    }
    public V getB() {
        return b;
    }
    public void showType() {
        System.out.println("Type of T is : " + a.getClass().getName());
        System.out.println("Type of V is : " + b.getClass().getName());
    }
    public static void main(String args[]) {
        Gen2<Integer, String> obj = new Gen2<Integer, String>();
        obj.setA(10);
        obj.setB("Welcome");
        System.out.println("Value : " + obj.getA());
        System.out.println("Value : " + obj.getB());
        obj.showType();
    }
}
```

Example: (Bounded Types)

```
class Gen3<T extends Number> {
    T num[];
    public Gen3(T num[]) {
        this.num = num;
    }
    public double average() {
        double sum = 0;
        for (int i = 0; i < num.length; i++) {
            sum += num[i].doubleValue();
        }
        return sum / num.length;
    }
    public static void main(String args[]) {
        Integer num[] = { 1, 2, 3, 4, 5, 6 };
        Gen3<Integer> obj = new Gen3<Integer>(num);
        System.out.println(obj.average());

        Double num1[] = { 1.1, 2.2, 3.3, 4.4, 5.4, 6.6 };
        Gen3<Double> obj1 = new Gen3<Double>(num1);
        System.out.println(obj1.average());
        /*
         * String str[] = {"Sunday","Monday","Tuesday"}; Gen3<String> obj3 = new
         * Gen3<String>(str);
         */
    }
}
```

Example: (Generic Method)

```
class Gen4 {
    public static <T, V extends T> boolean isIn(T num[], V num1) {
        for (int i = 0; i < num.length; i++) {
            if (num[i].equals(num1))
                return true;
        }
        return false;
    }

    public static void main(String args[]) {
        String str[] = { "One", "Two", "Three", "Five" };
        Integer num[] = { 1, 2, 3, 5 };
        if (!isIn(str, "Four"))
            System.out.println("Four not present");
        if (isIn(str, "Three"))
            System.out.println("Three  present");
        if (!isIn(num, 4))
            System.out.println("Four not present");
        if (isIn(num, 3))
            System.out.println("Three  present");
        if (isIn(num, "3"))
            System.out.println("Three  present");
    }
}
```

Example: (Generic Constructors)

```
class Gen5 {
    double data;
    public <T extends Number> Gen5(T obj) {
        data = obj.doubleValue();
    }
    public static void main(String args[]) {
        Gen5 obj1 = new Gen5(10);
        Gen5 obj2 = new Gen5(10l);
        Gen5 obj3 = new Gen5(10.5);
        Gen5 obj4 = new Gen5(10.6f);
    }
}
```

Example: (Generic Interfaces)

```
interface MyInterface<T> {
    void sayMessage(T t);
}
class MyClass implements MyInterface<String> {
    public void sayMessage(String t) {
        System.out.println(t);
    }

    public static void main(String args[]) {
        MyClass obj = new MyClass();
        obj.sayMessage("Hello World");
    }
}
```

Generic Restrictions

- Type parameters cannot be instantiated.
- No static member can use a type parameter declared by the enclosing class.
- You cannot instantiate an array whose element type is a type parameter.
- You cannot create an array of type-specific generic references.
- A generic class cannot extend Throwable.

Generic Assignment

1. Create a generic class Box<T> that can store and return an object of any type T. Implement methods:
   - void set(T item): To store an item.
   - T get(): To retrieve the item.

   Example:

   ```
   Box<Integer> intBox = new Box<>();
   intBox.set(10);
   System.out.println(intBox.get()); // Output: 10

   Box<String> strBox = new Box<>();
   strBox.set("Hello");
   System.out.println(strBox.get()); // Output: Hello
   ```

2. Create a generic class Pair<K, V> that stores a key-value pair of any type. Implement methods:
   - K getKey(): Returns the key.
   - V getValue(): Returns the value.

   Example:

   ```
   Pair<String, Integer> student = new Pair<>("Alice", 90);
   System.out.println(student.getKey());   // Output: Alice
   System.out.println(student.getValue()); // Output: 90

   Pair<Integer, String> idToName = new Pair<>(101, "John");
   System.out.println(idToName.getKey());   // Output: 101
   System.out.println(idToName.getValue()); // Output: John
   ```

3. Define a generic interface MinMax<T> with methods:
   - T min(T[] array): Returns the smallest element.
   - T max(T[] array): Returns the largest element.

   Implement this interface in a class and test it for an array of Integer.

4. Write a generic function swap(T[] array, int i, int j) that swaps two elements in an array. Test it for an Integer and String array.

   Example:

   ```
   Integer[] numbers = {1, 2, 3, 4};
   swap(numbers, 0, 2);
   System.out.println(Arrays.toString(numbers)); // Output: [3, 2, 1, 4]
   String[] words = {"A", "B", "C"};
   swap(words, 1, 2);
   System.out.println(Arrays.toString(words)); // Output: ["A", "C", "B"]
   ```

Day-7: Lambda

- Introducing Lambda Expressions
    - Lambda Expression Fundamentals
    - Functional Interfaces
    - Some Lambda Expression Examples
- Block Lambda Expressions
- Generic Functional Interfaces
- Passing Lambda Expressions as Arguments
- Method References
    - Method References to static Methods
    - Method References to Instance Methods
    - Constructor References
- Pre-Defined Functional Interface

Lambda

Added by JDK 8, lambda expressions (and their related features) significantly enhance Java because of two primary reasons.

- They added new syntax elements.
- They streamline the way that certain common constructs are implemented.

Introducing Lambda Expressions

- Java's implementations of lambda expressions are done using two constructs.
    1. The lambda expression.
    2. The functional interface.
- A **lambda expression** is, essentially, an anonymous (that is, unnamed) method. However, this method is not executed on its own. Instead, it is used to implement a method defined by a functional interface. Thus, a lambda expression results in a form of anonymous class. Lambda expressions are also commonly referred to as ***closures***.
- A **functional interface** is an interface that contains one and only one abstract method. Normally, this method specifies the intended purpose of the interface. Thus, a functional interface typically represents a single action. A functional interface is sometimes referred to as a **SAM** type, where SAM stands for **Single Abstract Method.**

Lambda Expression Fundamentals

- The lambda expression introduces a new syntax element and operator into the Java language.

```
<para> -> [<Single expression> or <Block of code>]
```

Example:

```
() -> 123.45
```

This lambda expression takes no parameters, thus the parameter list is empty. It returns the constant value 123.45. Therefore, it is similar to the following method:

```
double myMeth() { return 123.45; }
```

Example:

```
() -> Math.random() * 100
(n) -> (n % 2)==0
```

Functional Interfaces

- A functional interface is an interface that contains one and only one abstract method.

```
interface MyNumber {
	double getValue();
}
```

Example:

```
interface MyNumber {
	double getValue();
}
class LambdaDemo1 {
	public static void main(String args[]) {
		MyNumber myNum;
		myNum = () -> 123.45;
		System.out.println("A fixed value: " + myNum.getValue());
		myNum = () -> Math.random() * 100;
		System.out.println("A random value: " + myNum.getValue());
		System.out.println("Another random value: " + myNum.getValue());
		// myNum = () -> "123.03"; // Error!
	}
}
```

Example:

```
interface NumericTest {
	boolean test(int n);
}
public class LambdaDemo2 {
	public static void main(String args[]) {
		NumericTest isEven = (n) -> (n % 2) == 0;
		if (isEven.test(10))
			System.out.println("10 is even");
		if (!isEven.test(9))
			System.out.println("9 is not even");

		NumericTest isNonNeg = (n) -> n >= 0;
		if (isNonNeg.test(1))
			System.out.println("1 is non-negative");
		if (!isNonNeg.test(-1))
			System.out.println("-1 is negative");
	}
}
```

Example:

```
interface NumericTest2 {
	boolean test(int n, int d);
}
public class LambdaDemo3 {
	public static void main(String args[]) {
		NumericTest2 isFactor = (n, d) -> (n % d) == 0;
		if (isFactor.test(10, 2))
			System.out.println("2 is a factor of 10");
		if (!isFactor.test(10, 3))
			System.out.println("3 is not a factor of 10");
	}
}
```

Block Lambda Expressions

Example:

```
interface NumericFunc {
	int func(int n);
}
class BlockLambdaDemo {
	public static void main(String args[]) {
		// This block lambda computes the factorial of an int value.
		NumericFunc factorial = (n) -> {
			int result = 1;
			for (int i = 1; i <= n; i++)
				result = i * result;
			return result;
		};
		System.out.println("The factoral of 3 is " + factorial.func(3));
		System.out.println("The factoral of 5 is " + factorial.func(5));
	}
}
```

Example:

```
interface StringFunc {
	String func(String n);
}
class BlockLambdaDemo2 {
	public static void main(String args[]) {
	// This block lambda reverses the characters in a string.
		StringFunc reverse = (str) -> {
			String result = "";
			int i;
			for (i = str.length() - 1; i >= 0; i--)
				result += str.charAt(i);
			return result;
		};
		System.out.println("Lambda reversed is " + reverse.func("Lambda"));
		System.out.println("Expression reversed is " + reverse.func("Expression"));
	}
}
```

Generic Functional Interfaces

Example:

```
interface SomeFunc<T> {
	T func(T t);
}
class GenericFunctionalInterfaceDemo {
	public static void main(String args[]) {
		// Use a String-based version of SomeFunc.
		SomeFunc<String> reverse = (str) -> {
			String result = "";
```

```
            int i;
            for (i = str.length() - 1; i >= 0; i--)
                result += str.charAt(i);
            return result;
        };
        System.out.println("Lambda reversed is " + reverse.func("Lambda"));
        System.out.println("Expression reversed is " + reverse.func("Expression"));
        // Now, use an Integer-based version of SomeFunc.
        SomeFunc<Integer> factorial = (n) -> {
            int result = 1;
            for (int i = 1; i <= n; i++)
                result = i * result;
            return result;
        };
        System.out.println("The factoral of 3 is " + factorial.func(3));
        System.out.println("The factoral of 5 is " + factorial.func(5));
    }
}
```

Passing Lambda Expressions as Arguments

- Passing a lambda expression as an argument is a common use of lambdas.
- It is a very powerful use because it gives you a way to pass executable code as an argument to a method.

Example:

```
interface StringFunc {
    String func(String n);
}
public class LambdasAsArgumentsDemo {
    static String stringOp(StringFunc sf, String s) {
        return sf.func(s);
    }
    public static void main(String args[]) {
        String inStr = "Lambdas add power to Java";
        String outStr;
        System.out.println("Here is input string: " + inStr);
        outStr = stringOp((str) -> str.toUpperCase(), inStr);
        System.out.println("The string in uppercase: " + outStr);
        outStr = stringOp((str) -> {
            String result = "";
            int i;
            for (i = 0; i < str.length(); i++)
                if (str.charAt(i) != ' ')
                result += str.charAt(i);
            return result;
        }, inStr);
        System.out.println("The string with spaces removed: " + outStr);
        StringFunc reverse = (str) -> {
            String result = "";
            int i;
            for (i = str.length() - 1; i >= 0; i--)
```

```
                result += str.charAt(i);
            return result;
        };
        System.out.println("The string reversed: " + stringOp(reverse, inStr));
    }
}
```

Method References

Method References to static Methods

- To create a static method reference, use this general syntax:

ClassName::methodName

```
interface StringFunc {
    String func(String n);
}
class MyStringOps {
    static String strReverse(String str) {
        String result = "";
        int i;
        for (i = str.length() - 1; i >= 0; i--)
            result += str.charAt(i);
        return result;
    }
}
class MethodRefDemo {
    static String stringOp(StringFunc sf, String s) {
        return sf.func(s);
    }
    public static void main(String args[]) {
        String inStr = "Lambdas add power to Java";
        String outStr;
        outStr = stringOp(MyStringOps::strReverse, inStr);
        System.out.println("Original string: " + inStr);
        System.out.println("String reversed: " + outStr);
    }
}
```

Method References to Instance Methods

- To pass a reference to an instance method on a specific object, use this basic syntax:

objRef::methodName

Example:

```
interface StringFunc {
    String func(String n);
}
class MyStringOps {
    String strReverse(String str) {
        String result = "";
        int i;
        for (i = str.length() - 1; i >= 0; i--)
            result += str.charAt(i);
```

```
                return result;
        }
}
class MethodRefDemo2 {
        static String stringOp(StringFunc sf, String s) {
                return sf.func(s);
        }
        public static void main(String args[]) {
                String inStr = "Lambdas add power to Java";
                String outStr;
                MyStringOps strOps = new MyStringOps();
                outStr = stringOp(strOps::strReverse, inStr);
                System.out.println("Original string: " + inStr);
                System.out.println("String reversed: " + outStr);
        }
}
```

Constructor References

- Similar to the way that you can create references to methods, you can create references to constructors.

classname::new

```
interface MyFunc {
        MyClass func(int n);
}
class MyClass {
        private int val;
        MyClass(int v) {
                val = v;
        }
        MyClass() {
                val = 0;
        }
        int getVal() {
                return val;
        };
}
class ConstructorRefDemo {
        public static void main(String args[]) {
                MyFunc myClassCons = MyClass::new;
                MyClass mc = myClassCons.func(100);
                System.out.println("val in mc is " + mc.getVal());
        }
}
```

Lamdba Assignment

1. Lambda Expression Fundamentals & Functional Interfaces

Create a Functional Interface MathOperation with a method int operate(int a, int b).
Implement it using a lambda expression for addition, subtraction, multiplication, and division.

2. **Block Lambda Expressions**

Write a lambda expression that takes an integer and returns its factorial using a block lambda expression. Use a functional interface Factorial with a method long compute(int n).

3. **Method Reference**

Create a String library class that contains the following methods:

- public static String toTitleCase(String str)
- public static String toCamelCase(String str)
- public static String removeSpaces(String str)
- public static int countVowels(String str)
- public static int countConsonants(String str)
- public static int countAlphabets(String str)
- public static int countWords(String str)

4. Create 2 functional interfaces for String library and provide definition for Single Abstract Method (SAM).
5. Create a menu driven application with following options:
   a. Convert string to title case.
   b. Convert string to camel case.
   c. Remove spaces from string.
   d. Count number of vowels in a string.
   e. Count number of consonants in a string
   f. Count number of alphabets in a string
   g. Count number of words in a string
   h. Exit.

<u>Predefined Functional Interfaces</u>

- Java 8 has provided multiple Predefined (Built-in) Functional Interfaces to make our programming easier.
- When we use predefined functional interfaces, code becomes more readable and maintainable.
- These interfaces comes from java.util.function package.

Predicate<T>

```
public interface Predicate<T> {
   boolean test(T t);
}
```

We can use Predicate<T> to implement some conditional checks.

```
Predicate<Integer> p = (i) -> (i > -10) && (i < 10);
System.out.println(p.test(9));
```

BiPredicate<T, U>

```
interface BiPredicate<T, U> {
   boolean test(T t, U u)
}
```

BiPredicate<T, U> is same as Predicate<T> except that it has two input parameters.

```
BiPredicate<Integer,Integer> bp = (i, j) -> (i + j) %2 == 0;
System.out.println(bp.test(24, 34));
```

Function<T, R>

```
interface Function<T,R> {
   R apply(T t);
}
```

Function<T, R> is used to perform some operation & returns some result.

```
Function<String, Integer> f = s -> s.length();
System.out.println(f.apply("I am happy happy now"));
```

BiFunction<T, U, R>

```
interface BiFunction<T, U, R> {
    R apply(T t, U u);
}
```

BiFunction<T, U, R> is same as Function<T, R> except that it has two input parameters.

```
BiFunction<Integer,Integer,Integer> bf = (i, j) -> i + j;
System.out.println(bf.apply(24, 4));
```

Consumer<T>

```
interface Consumer<T> {
   void accept(T t);
}
```

Consumer<T> is used when we have to provide some input parameter, perform certain operation, but don’t need to return anything.

```
Consumer<String> c = s -> System.out.println(s);
c.accept("I consume data but don't return anything");
```

BiConsumer<T, U>

```
interface BiConsumer<T, U> {
   void accept(T t, U u);
}
```

BiConsumer<T> is same as Consumer<T> except that it has two input parameters.

```
BiConsumer<String,String> bc =  (s1, s2) -> System.out.println(s1+s2);
bc.accept("Bi","Consumer");
```

Supplier<R>

```
interface Supplier<R>{
    R get();
}
```

Supplier<R> doesn’t take any input and it always returns some object.

```
Supplier<String> otps = () -> {
     String otp = "";
     for (int i = 1; i <= 4; i++) {
        otp = otp + (int) (Math.random() * 10);
     }
   return otp;
};
System.out.println(otps.get());
System.out.println(otps.get());
```

Day-7: Annotations (Metadata)

- Annotations was introduced with jdk5
- It enables you to embed supplemental information into a source file.
- It does not change the actions of a program; it leaves the semantics of a program unchanged.
- Supplement information can be used by various tools during both development and deployment. For example source code generator
- All annotation types automatically extend the Annotation interface.
- Any type of declaration can have an annotation associated with it. For example, classes, methods, fields, parameters, and enum constants can be annotated. Even an annotation can be annotated.

Annotation Basics

An annotation is created through a mechanism based on the interface.

```
// A simple annotation type.
@interface MyAnno {
    String str();
    int val();
}
```

Retention Policy

- A retention policy determines at what point an annotation is discarded.
- Java defines three policies, which are within the java.lang.annotation.RetentionPolicy enumeration. They are SOURCE, CLASS, and RUNTIME.
- An annotation with a retention policy of SOURCE is retained only in the source file and is discarded during compilation.
- An annotation with a retention policy of CLASS is stored in the .class file during compilation. It is not available through the JVM during run time. (default)
- An annotation with a retention policy of RUNTIME is stored in the .class file during compilation and is available through the JVM during run time.
- A retention policy is specified for an annotation by using one of Java’s built-in annotations: @Retention.
  @Retention(retention-policy)

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
    String str();
    int val();
}
```

Example:

```
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
    String str();
    int val();
}
class AnnotationExample1 {
    @MyAnno(str = "Annotation Example", val = 100)
    public static void myMeth() {
        AnnotationExample1 ob = new AnnotationExample1();
        try {
            Class<?> c = ob.getClass();
            Method m = c.getMethod("myMeth");
            MyAnno anno = m.getAnnotation(MyAnno.class);
            System.out.println(anno.str() + " " + anno.val());
        } catch (NoSuchMethodException exc) {
            System.out.println("Method Not Found.");
        }
    }

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

Example:

```
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
    String str();
    int val();
}

class AnnotationExample2 {
    @MyAnno(str = "Two Parameters", val = 19)
    public static void myMeth(String str, int i) {
        AnnotationExample2 ob = new AnnotationExample2();
        try {
            Class<?> c = ob.getClass();
            Method m = c.getMethod("myMeth", String.class, int.class);
            MyAnno anno = m.getAnnotation(MyAnno.class);
            System.out.println(anno.str() + " " + anno.val());
        } catch (NoSuchMethodException exc) {
            System.out.println("Method Not Found.");
        }
    }

    public static void main(String args[]) {
        myMeth("test", 10);
    }
}
```

Example: (Obtaining All Annotations)

```
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
```

```
@interface MyAnno {
    String str();
    int val();
}

@Retention(RetentionPolicy.RUNTIME)
@interface What {
    String description();
}

@What(description = "An annotation test class")
@MyAnno(str = "Meta2", val = 99)
class AnnotationExample3 {
    @What(description = "An annotation test method")
    @MyAnno(str = "Testing", val = 100)
    public static void myMeth() {
          AnnotationExample3 ob = new AnnotationExample3();
          try {
                    Annotation annos[] = ob.getClass().getAnnotations();
                    System.out.println("All annotations for AnnotationExample3:");
                    for(Annotation a : annos)
                              System.out.println(a);
                    System.out.println();

                    Method m = ob.getClass( ).getMethod("myMeth");
                    annos = m.getAnnotations();
                    System.out.println("All annotations for myMeth:");
                    for(Annotation a : annos)
                              System.out.println(a);
          } catch (NoSuchMethodException exc) {
                    System.out.println("Method Not Found.");
          }
    }
    public static void main(String args[]) {
          myMeth();
    }
}
```

Example: (Using default values)

```
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
    String str() default "Testing";
    int val() default 9000;
}

class AnnotationExample4 {
    @MyAnno()
    public static void myMeth() {
        AnnotationExample4 ob = new AnnotationExample4();
        try {
                Class<?> c = ob.getClass();
                Method m = c.getMethod("myMeth");
                MyAnno anno = m.getAnnotation(MyAnno.class);
                System.out.println(anno.str() + " " + anno.val());
        } catch (NoSuchMethodException exc) {
                System.out.println("Method Not Found.");
        }
    }

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

Example : (Marker)

```
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@interface MyMarker { }

class Marker {

    @MyMarker
    public static void myMeth() {
        Marker ob = new Marker();
        try {
                Method m = ob.getClass().getMethod("myMeth");
                if(m.isAnnotationPresent(MyMarker.class))
                        System.out.println("MyMarker is present.");
        } catch (NoSuchMethodException exc) {
                        System.out.println("Method Not Found.");
        }
    }

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

Example: (Single Member)

```
import java.lang.annotation.*;
import java.lang.reflect.*;
```

```
@Retention(RetentionPolicy.RUNTIME)
@interface MySingle {
    int value();
}

class Single {

    @MySingle(100)
    public static void myMeth() {
        Single ob = new Single();
        try {
                Method m = ob.getClass().getMethod("myMeth");
                MySingle anno = m.getAnnotation(MySingle.class);
                System.out.println(anno.value());
        } catch (NoSuchMethodException exc) {
                System.out.println("Method Not Found.");
        }
    }

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

Restrictions on Annotations

- No annotation can inherit another.
- All methods declared by an annotation must be without parameters and they must return one of the following:
  - A primitive type, such as int or double
  - An object of type String or Class
  - An enum type
  - Another annotation type
  - An array of one of the preceding types
- Annotations cannot be generic. In other words, they cannot take type parameters.
- annotation methods cannot specify a throws clause.

Java Annotations Assignment

Part 1: Marker Annotation

1. **Create a marker annotation** @ImportantTask and apply it to a method in a class.
    - Write a program to check if the method has this annotation using reflection.

Part 2: Single-Member Annotation

2. **Create a single-member annotation** @Author that takes a String value (author's name).
    - Apply it to a class and display the author name using reflection.

Part 3: Custom Annotation with Multiple Values

3. Define an annotation @Info with attributes version (double) and status (String).
    - Apply it to a class and retrieve its values.