MODULE 1

Enumerations, Autoboxing and Annotations

Enumeration in Java

- **Definition**: An enumeration is a data type that consists of a fixed set of constants. It can represent days of the week, directions, etc., and has been available since JDK 1.5.
- **Characteristics**:
  - Constants are public, static, and final by default.
  - Enumeration can have constructors, methods, and instance variables, but you do not instantiate an enum using new.
  - Enumerations are declared similarly to primitive variables.

Defining and Using Enumerations

- **Syntax**:

```
enum Subject {
   JAVA, CPP, C, DBMS
}
```

- **Usage**:
  - Declare an enum variable: Subject sub;
  - Assign a value: sub = Subject.JAVA;
  - Compare values using ==: if(sub == Subject.JAVA) { ... }

Example Programs

1. **Simple Enumeration**:

```
enum WeekDays {
   sun, mon, tues, wed, thurs, fri, sat
}
class Test {
   public static void main(String args[]) {
      WeekDays wk = WeekDays.sun;
      System.out.println("Today is " + wk);
   }
}
```

Output: Today is sun

2. **Enumeration with Switch Statement**:

```
enum Restaurants {
   DOMINOS, KFC, PIZZAHUT, PANINOS, BURGERKING
}
class Test {
   public static void main(String args[]) {
      Restaurants r = Restaurants.PANINOS;
      switch(r) {
         case DOMINOS:
            System.out.println("I AM " + r.DOMINOS);
            break;
         case KFC:
            System.out.println("I AM " + r.KFC);
            break;
         case PIZZAHUT:
            System.out.println("I AM " + r.PIZZAHUT);
            break;
         case PANINOS:
            System.out.println("I AM " + r.PANINOS);
            break;
         case BURGERKING:
            System.out.println("I AM " + r.BURGERKING);
            break;
      }
   }
}
```

Output: I AM PANINOS

values() and valueOf() Methods

- **values() Method**:
    - Added by the Java compiler to an enum.
    - Returns an array containing all the values of the enum.
    - **General Form**:

public static enum-type[] values()

- **valueOf() Method**:
    - Returns the enumeration constant whose value is equal to the string passed as an argument.
    - **General Form**:

public static enum-type valueOf(String str)

Example Program Using values() and valueOf() Methods

java

Copy code

```
enum Restaurants {
   DOMINOS, KFC, PIZZAHUT, PANINOS, BURGERKING
}


class Test {
   public static void main(String args[]) {
      Restaurants r;
      System.out.println("All constants of enum type Restaurants are:");
      Restaurants rArray[] = Restaurants.values(); // Returns an array of constants of type
Restaurants


      for(Restaurants a : rArray) { // Using foreach loop
         System.out.println(a);
      }
      r = Restaurants.valueOf("DOMINOS");
      System.out.println("I AM " + r);
   }
```

}

Output:

All constants of enum type Restaurants are:

DOMINOS

KFC

PIZZAHUT

PANINOS

BURGERKING

I AM DOMINOS

Points to Remember About Enumerations

1. Enumerations are of class type and have all the capabilities of a Java class.
2. Enumerations can have constructors, instance variables, methods, and can even implement interfaces.
3. Enumerations are not instantiated using the new keyword.
4. All enumerations by default inherit the java.lang.Enum class.
5. An enum may implement multiple interfaces but cannot extend any class because it internally extends the Enum class.

Java Enumerations as Class Types

- **Class Type**:
    - Each enum constant is an object of its enumeration type.
    - Constructors for an enum are called when each constant is created.
    - Each constant has its own copy of any instance variables defined by the enum.

Example: Enumeration with Constructor, Instance Variable, and Method

```
enum Apple2 {
   Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);


   // Variable
   int price;


   // Constructor
   Apple2(int p) {
```

```
        price = p;
    }

    // Method
    int getPrice() {
        return price;
    }
}

public class EnumConstructor {
    public static void main(String[] args) {
        Apple2 ap;

        // Display price of Winesap
        System.out.println("Winesap costs " + Apple2.Winesap.getPrice() + " cents.\n");

        // Display price of GoldenDel
        System.out.println(Apple2.GoldenDel.price);

        // Display all apples and prices
        System.out.println("All apple prices:");
        for (Apple2 a : Apple2.values()) {
            System.out.println(a + " costs " + a.getPrice() + " cents.");
        }
    }
}
```

Output:

Winesap costs 15 cents.

9

All apple prices:

Jonathan costs 10 cents.

GoldenDel costs 9 cents.

RedDel costs 12 cents.

Winesap costs 15 cents.

Cortland costs 8 cents.

Enumerations Inherit Enum

- **Inheritance**:
  - All enumerations inherit from java.lang.Enum.
  - Methods provided by this class include ordinal(), compareTo(), and equals().
- **ordinal() Method**:
  - Returns the ordinal value of the invoking constant.
  - **General Form**: final int ordinal()
  - Ordinal values start at zero.
- **compareTo() Method**:
  - Compares the ordinal value of two constants.
  - **General Form**: final int compareTo(enum-type e)
  - Returns a negative value, zero, or a positive value depending on the comparison.

Example: Using ordinal(), compareTo(), and equals()

```
enum Apple5 {
   Jonathan, GoldenDel, RedDel, Winesap, Cortland
}

public class EnumOrdinal {
   public static void main(String[] args) {
      Apple5 ap, ap2, ap3;

      // Obtain all ordinal values using ordinal()
      System.out.println("Here are all apple constants and their ordinal values:");
      for (Apple5 a : Apple5.values()) {
         System.out.println(a + " " + a.ordinal());
```

```
        }

        ap = Apple5.RedDel;
        ap2 = Apple5.GoldenDel;
        ap3 = Apple5.RedDel;

        // Demonstrate compareTo() and equals()
        System.out.println();
        if (ap.compareTo(ap2) < 0) {
          System.out.println(ap + " comes before " + ap2);
        }
        if (ap.compareTo(ap2) > 0) {
          System.out.println(ap2 + " comes before " + ap);
        }
        if (ap.compareTo(ap3) == 0) {
          System.out.println(ap + " equals " + ap3);
        }

        System.out.println();
        if (ap.equals(ap2)) {
          System.out.println("Error!");
        }
        if (ap.equals(ap3)) {
          System.out.println(ap + " equals " + ap3);
        }
    }
}
```

Output:

Here are all apple constants and their ordinal values:

Jonathan 0

GoldenDel 1

RedDel 2

Winesap 3

Cortland 4

RedDel comes before GoldenDel

RedDel equals RedDel

RedDel equals RedDel

Java Type Wrappers

Java provides type wrappers to encapsulate primitive types within objects. This is useful for scenarios where you need to use primitive types as objects, such as when working with Java Collections or other object-oriented frameworks.

1. **Primitive Types and Performance**:
    - Java uses primitive types (e.g., int, double, float) for performance reasons.
2. **Need for Object Representation**:
    - Many data structures and APIs in Java require objects, not primitives. Hence, type wrappers are necessary.
3. **Type Wrappers**:
    - Java provides wrapper classes for each primitive type, allowing them to be treated as objects.

Wrapper Classes:

1. **Character**:
    - **Purpose**: Encapsulates the primitive type char.
    - **Constructor**: Character(char ch)
2. **Boolean**:
    - **Purpose**: Encapsulates the primitive type boolean.
    - **Constructor**: Boolean(boolean boolValue)
3. **Numeric Type Wrappers**:
    - **Byte**: Byte
    - **Short**: Short
    - **Integer**: Integer
    - **Long**: Long
    - **Float**: Float

- **Double**: Double

Boxing and Unboxing

Boxing and unboxing in Java deal with the conversion between primitive types and their corresponding wrapper classes. This is particularly useful for situations where you need to work with objects instead of primitive types.

Key Concepts:

1. **Boxing**: Encapsulating a primitive value within an object.
2. **Unboxing**: Extracting the primitive value from its wrapper object.

Example of Boxing and Unboxing:

```
class Wrap {
   public static void main(String args[]) {
      Integer iOb = new Integer(100); // Boxing
      int i = iOb.intValue(); // Unboxing
      System.out.println(i + " " + iOb); // displays 100 100
   }
}
```

Explanation:

- **Boxing**: Integer iOb = new Integer(100); - Encapsulates the primitive value 100 into an Integer object.
- **Unboxing**: int i = iOb.intValue(); - Extracts the primitive value from the Integer object.

Autoboxing and Unboxing

Java 5 introduced autoboxing and auto-unboxing, simplifying the process by allowing automatic conversion between primitives and their wrapper objects.

Example of Autoboxing and Unboxing:

```
class Test {
   public static void main(String[] args) {
      Integer iob = 100; // Autoboxing
      int i = iob; // Auto-unboxing
      System.out.println(i + " " + iob); // displays 100 100

      Character cob = 'a'; // Autoboxing
```

```
        char ch = cob; // Auto-unboxing
        System.out.println(cob + " " + ch); // displays a a
    }
}
```

Output:

100 100

a a

Autoboxing/Unboxing in Expressions

When using wrapper objects in expressions, Java automatically unboxes and reboxes values as needed.

Example:

```
class Test {
    public static void main(String[] args) {
        Integer iOb;
        iOb = 100; // Autoboxing
        ++iOb; // Unboxing, incrementing, and reboxing
        System.out.println(iOb); // displays 101
    }
}
```

Benefits of Autoboxing/Unboxing

1. **Interchangeability**: Allows primitive types and wrapper class objects to be used interchangeably.
2. **No Explicit Casting**: Eliminates the need for explicit type casting.
3. **Error Prevention**: Helps prevent errors that can occur during manual boxing/unboxing.

Mixing Different Types of Numeric Objects

Autoboxing and unboxing allow mixing different types of numeric objects in expressions, automatically handling conversions.

Example:

```
class Test {
    public static void main(String args[]) {
        Integer i = 35;
```

```
        Double d = 33.3;
        d = d + i; // Auto-unboxing, addition, and autoboxing
        System.out.println("Value of d is " + d); // displays Value of d is 68.3
    }
}
```

Output:

Value of d is 68.3

Error Prevention with Autoboxing/Unboxing

Autoboxing and unboxing help prevent errors that can occur with manual boxing/unboxing, such as type mismatches.

Example of Error with Manual Unboxing:

```
class UnboxingError {
    public static void main(String args[]) {
        Integer iOb = 1000; // autobox the value 1000
        int i = iOb.byteValue(); // manually unbox as byte
        System.out.println(i); // does not display 1000 !
    }
}
```

Output:

-24

Explanation:

- The value 1000 is manually unboxed using byteValue(), causing truncation and resulting in an incorrect value of -24.
- Autoboxing/unboxing avoids such errors by ensuring the proper type conversion.

Annotations / Metadata in Java

Annotations in Java provide a way to add metadata to your code, allowing tools, debuggers, and applications to understand and utilize this information. This metadata can help in analyzing, documenting, and controlling the behavior of the code.

Key Points:

1. **Definition**:
    - An annotation is a form of metadata that can be applied to various elements of Java source code, including classes, methods, variables, parameters, and packages.
2. **Purpose**:
    - Annotations are used to add additional information to the code, which can be utilized by tools and frameworks during compilation or at runtime.
3. **Compilation and Reflection**:
    - Annotations are compiled into bytecode and can be accessed using reflection. This allows the program to query the metadata information and perform actions accordingly.

Example: Understanding Metadata

Consider the example of a class declared as final:

```
public final class MyFinalClass {
   // other class members
}
```

- **Metadata**: The final keyword in the class declaration is a metadata indicator.
- **Purpose**: It tells the compiler and JVM that this class cannot be subclassed.

Metadata in Java Context

- **Data About Data**: Metadata provides additional information about the code, helping the JVM and developers understand and use the code effectively.
- **Example**: The final keyword in public final class MyFinalClass adds a flag indicating that the class is final

Built-In Annotations in Java

Java provides several built-in annotations that are used to convey metadata about the code, which can be utilized by the compiler and runtime environment. These annotations help in various aspects such as code documentation, compiler warnings, and specifying how custom annotations should be used.

Built-In Java Annotations for Code

1. **@Override**
    - **Purpose**: Ensures that a method in a subclass is overriding a method in its superclass. If not, it results in a compile-time error.

- **Example**:

```
class Animal {
    void eatSomething() {
        System.out.println("eating something");
    }
}

class Dog extends Animal {
    @Override
    void eatSomething() {
        System.out.println("eating foods");
    }
}

public class AnnotationDemo1 {
    public static void main(String[] args) {
        Animal a = new Dog();
        a.eatSomething(); // Output: eating foods
    }
}
```

2. **@SuppressWarnings**
   - **Purpose**: Suppresses specified compiler warnings. Useful for ignoring warnings about deprecated APIs or unchecked operations.
   - **Example**:

```
import java.util.*;

class AnnotationDemo2 {
    @SuppressWarnings("unchecked")
    public static void main(String[] args) {
        ArrayList list = new ArrayList();
        list.add("a");
```

```
        list.add("b");
        list.add("c");
        for (Object obj : list) {
            System.out.println(obj);
        }
    }
}
```

3. **@Deprecated**
    - **Purpose**: Marks a method or class as deprecated, meaning it is no longer recommended for use and may be removed in future versions.
    - **Example**:

```
class A {
    void m() {
        System.out.println("hello m");
    }

    @Deprecated
    void n() {
        System.out.println("hello n");
    }
}

class AnnotationDemo3 {
    public static void main(String[] args) {
        A a = new A();
        a.n(); // Output: hello n
    }
}
```

Built-In Java Annotations for Other Annotations

1. **@Retention**

- **Purpose**: Specifies how the annotation should be stored and accessed. It can be retained at runtime, compile-time, or only in the source code.
- **Example**:

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME)
public @interface MyCustomAnnotation {
   // Some code
}
```

2. **@Documented**
   - **Purpose**: Indicates that the annotation should be included in the Javadoc generated for the annotated element.
   - **Example**:

```
import java.lang.annotation.Documented;

@Documented
public @interface MyCustomAnnotation {
   // Some code
}
```

3. **@Target**
   - **Purpose**: Specifies the kinds of elements an annotation can be applied to, such as classes, methods, fields, etc.
   - **Example**:

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

@Target({ElementType.TYPE, ElementType.METHOD})
public @interface MyCustomAnnotation {
   // Some code
}
```

4. **@Inherited**
    - **Purpose**: Indicates that if an annotation is applied to a class, its subclasses will inherit the annotation. Useful for class-level annotations that should be inherited by subclasses.
    - **Example**:

```
import java.lang.annotation.Inherited;

@Inherited
public @interface MyCustomAnnotation {
   // Some code
}


@MyCustomAnnotation
class MyParentClass {
   // Some code
}


class MyChildClass extends MyParentClass {
   // This class will inherit the MyCustomAnnotation
}
```

Summary

- **Annotations Applied to Code**:
    - @Override: Ensures a method overrides a superclass method.
    - @SuppressWarnings: Suppresses specific compiler warnings.
    - @Deprecated: Marks elements as deprecated.
- **Annotations Applied to Other Annotations**:
    - @Retention: Specifies the retention policy of an annotation.
    - @Documented: Indicates that the annotation should be included in Javadoc.
    - @Target: Specifies where the annotation can be applied.
    - @Inherited: Indicates that the annotation is inherited by subclasses.

Annotations provide a powerful mechanism for adding metadata to Java code, helping with documentation, code analysis, and runtime behavior.

Custom/User-Defined Annotations in Java

Custom annotations allow you to create your own metadata to annotate classes, methods, fields, or other elements of your code. This can be particularly useful for code analysis, documentation, and even runtime behavior.

Creating Custom Annotations

To define a custom annotation in Java, follow these steps:

1. **Define the Annotation**: Use the @interface keyword to declare a custom annotation.
2. **Annotation Elements**: Define methods within the annotation to specify elements. These methods act as elements of the annotation type. They must not have parameters or a throws clause.
3. **Return Types**: The return types for annotation elements are restricted to:
    - Primitives (int, double, etc.)
    - String
    - Class
    - Enums
    - Other annotations
    - Arrays of the above types
4. **Default Values**: You can provide default values for annotation elements.

Types of Annotations

1. **Marker Annotation**
    - **Definition**: An annotation with no methods.
    - **Example**:

```
@interface MyMarkerAnnotation {
}
```

2. **Single-Value Annotation**
    - **Definition**: An annotation with one method.
    - **Example**:

java

Copy code

```
@interface MySingleValueAnnotation {
   int value() default 0;
}
```

3. **Multi-Value Annotation**
    - **Definition**: An annotation with more than one method.
    - **Example**:

```
@interface MyMultiValueAnnotation {
   int value1() default 1;
   String value2() default "";
   String value3() default "xyz";
}
```

Example of Custom Annotations

Here's how you can define and use a custom annotation:

1. **Define the Annotation**:

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import java.lang.annotation.ElementType;
import java.lang.annotation.Documented;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@Documented
@interface MyAnnotation2 {
   int value() default 2;
   String name() default "cse";
}
```

2. **Apply the Annotation**:

```
class Hello {
   @MyAnnotation2(value=4, name="ise")
   public void sayHello() {
      // Some code
   }
```

```
}
```

3. **Retrieve and Use the Annotation**:

```
import java.lang.reflect.Method;
public class CustomAnnotation {
   public static void main(String[] args) {
      try {
         Hello h = new Hello();
         // Get Class object representing the class
         Class<?> c = h.getClass();
         // Get Method object representing the method
         Method m = c.getMethod("sayHello");
         // Get the annotation from the method
         MyAnnotation2 anno = m.getAnnotation(MyAnnotation2.class);
         // Display the values
         System.out.println("Value: " + anno.value());
         System.out.println("Name: " + anno.name());
      } catch (Exception e) {
         System.out.println("No such method exception: " + e.getMessage());
      }
   }
}
```

Restrictions on Annotations

- **No Inheritance**: Annotations cannot inherit from other annotations.
- **Method Constraints**: Methods in annotations must:
    - Not have parameters.
    - Not have a throws clause.
    - Return one of the following:
        - A primitive type (e.g., int, double).
        - An object of type String or Class.
        - An enum type.

- Another annotation type.
- An array of one of the preceding types.

Custom annotations provide a flexible way to add metadata to Java code, which can be utilized at compile-time or runtime to affect program behavior, document code, or enforce certain constraints.