CE204 Object-Oriented Programming

Week-3 (OOP with Java-III)

Spring Semester, 2021-2022

Download DOC, SLIDE, PPTX

OOP with Java-III

Outline (1)

- Defining an Interface in Java
- Implementing an Interface in Java
- Nested Interfaces in Java
- Variables in Java Interfaces
- Extending an Interface in java

Outline (2)

- Advantages of Interface in Java
- default methods in Java Interfaces
- private and static Methods in Interface
- Java Reflection
- Java Wrapper Classes
- Java Lambda Expressions

Defining an Interface in Java

Defining an Interface in Java

- In java, an interface is similar to a class,
  - but it contains abstract methods and static final variables only.
  - The interface in Java is another mechanism to achieve abstraction.
- We may think of an interface as a completely abstract class.
  - None of the methods in the interface has an implementation,
  - and all the variables in the interface are constants.
- All the methods of an interface,
  - implemented by the class that implements it.
- The interface in java enables java to support multiple-inheritance.
  - An interface may extend only one interface,
  - but a class may implement any number of interfaces.

Defining an Interface in Java

- An interface is a container of abstract methods and static final variables.
- An interface, implemented by a class. (class implements interface).
- An interface may extend another interface. (Interface extends Interface).
- An interface never implements another interface, or class.
- A class may implement any number of interfaces.
- We can not instantiate an interface.
- Specifying the keyword abstract for interface methods is optional, it automatically added.
- All the members of an interface are public by default.

Defining an Interface in Java

- Defining an interface is similar to that of a class. We use the keyword interface to define an interface.
- All the members of an interface are public by default. The following is the syntax for defining an interface.

```
interface InterfaceName{
    ...
    members declaration;
    ...
}
```

Defining an Interface in Java

- In the example we defined an interface HumanInterfaceExample that contains two abstract methods learn(), work() and one constant duration.

```
interface HumanInterfaceExample {

        void learn(String str);
        void work();

        int duration = 10;

}
```

Defining an Interface in Java Example-1

- Every interface in Java is auto-completed by the compiler. For example, in the above example code,
- no member is defined as public, but all are public automatically.
- The above code automatically converted as follows.

```
interface HumanInterfaceExample {

        public abstract void learn(String str);
        public abstract void work();

        public static final int duration = 10;

}
```

Implementing an Interface in Java

Implementing an Interface in Java

- In java, an interface is implemented by a class.
- The class that implements an interface must provide code for all the methods defined in the interface, otherwise,
  - it must be defined as an abstract class.
- The class uses a keyword `implements` to implement an interface.
- A class can implement any number of interfaces.
- When a class wants to implement more than one interface,
  - we use the implements keyword is followed by a comma-separated list of the interfaces implemented by the class.

Implementing an Interface in Java

- The following is the syntax for defineing a class that implements an interface.

```
class className implements InterfaceName{
    ...
    boby-of-the-class
    ...
}
```

Implementing an Interface in Java Example-1

```
interface Human {

        void learn(String str);
        void work();

        int duration = 10;

}
```

Implementing an Interface in Java Example-1

```
class Programmer implements Human{
        public void learn(String str) {
                System.out.println("Learn using " + str);
        }
        public void work() {
                System.out.println("Develop applications");
        }
}
```

Implementing an Interface in Java Example-1

```
public class HumanTest {

        public static void main(String[] args) {
                Programmer trainee = new Programmer();
                trainee.learn("coding");
                trainee.work();
        }
}
```

Implementing an Interface in Java Example-1

- In the example we defined an interface
  - Human that contains two abstract methods
    - learn(),
    - work() and one constant duration.
- The class Programmer implements the interface.
  - As it implementing the Human interface it must provide the body of all the methods those defined in the Human interface.

Implementing an Interface in Java Example-2

```
interface Polygon {
  void getArea(int length, int breadth);
}
```

```
// implement the Polygon interface
class Rectangle implements Polygon {

  // implementation of abstract method
  public void getArea(int length, int breadth) {
    System.out.println("The area of the rectangle is " + (length * breadth));
  }
}
```

Implementing an Interface in Java Example-2

```
class Main {
  public static void main(String[] args) {
    Rectangle r1 = new Rectangle();
    r1.getArea(5, 6);
  }
}
```

Implementing an Interface in Java Example-3

```
// create an interface
interface Language {
  void getName(String name);
}
```

```
// class implements interface
class ProgrammingLanguage implements Language {

  // implementation of abstract method
  public void getName(String name) {
    System.out.println("Programming Language: " + name);
  }
}
```

Implementing an Interface in Java Example-3

```
class Main {
  public static void main(String[] args) {
    ProgrammingLanguage language = new ProgrammingLanguage();
    language.getName("Java");
  }
}
```

Implementing multiple Interfaces

- When a class wants to implement more than one interface,
- we use the implements keyword is followed by a comma-separated list of the interfaces implemented by the class.

Implementing multiple Interfaces

- The following is the syntax for defineing a class that implements multiple interfaces.

```
class className implements InterfaceName1, InterfaceName2, ...{
    ...
    boby-of-the-class
    ...
}
```

Implementing multiple Interfaces Example-1

- In the example we defined a class that implements multiple interfaces.

```
interface Human {
        void learn(String str);
        void work();
}
```

```
interface Recruitment {
        boolean screening(int score);
        boolean interview(boolean selected);
}
```

Implementing multiple Interfaces Example-1

```
class Programmer implements Human, Recruitment {
        public void learn(String str) {
                System.out.println("Learn using " + str);
        }
        public boolean screening(int score) {
                System.out.println("Attend screening test");
                int thresold = 20;
                if(score > thresold)
                        return true;
                return false;
        }
        public boolean interview(boolean selected) {
                System.out.println("Attend interview");
                if(selected)
                        return true;
                return false;
        }
        public void work() {
                System.out.println("Develop applications");
        }
}
```

Implementing multiple Interfaces Example-1

```
public class HumanTest {
        public static void main(String[] args) {
                Programmer trainee = new Programmer();
                trainee.learn("Coding");
                trainee.screening(30);
                trainee.interview(true);
                trainee.work();
        }
}
```

- In the example, two interfaces `Human` and `Recruitment`, and a class `Programmer` implements both the interfaces.

Implementing multiple Interfaces Example-2

```
interface A {
  // members of A
}
```

```
interface B {
  // members of B
}
```

```
class C implements A, B {
  // abstract members of A
  // abstract members of B
}
```

Nested Interfaces in Java

Nested Interfaces in Java

- In java, an interface may be defined inside another interface,
  - and also inside a class.
- The interface that defined inside another interface or a class is konwn as nested interface.
  - The nested interface is also refered as inner interface.

Nested Interfaces in Java

- The nested interface declared within an interface is public by default.
- The nested interface declared within a class can be with any access modifier.
- Every nested interface is static by default.

Nested Interfaces in Java

- The nested interface cannot be accessed directly.
  - We can only access the nested interface by using outer interface or outer class name followed by dot(.), followed by the nested interface name.

Nested interface inside another interface Example

- The nested interface that defined inside another interface must be accessed as `OuterInterface.InnerInterface` .

```
interface OuterInterface{
        void outerMethod();

        interface InnerInterface{
                void innerMethod();
        }
}
```

Nested interface inside another interface Example

```
class OnlyOuter implements OuterInterface{
        public void outerMethod() {
                System.out.println("This is OuterInterface method");
        }
}
```

```
class OnlyInner implements OuterInterface.InnerInterface{
        public void innerMethod() {
                System.out.println("This is InnerInterface method");
        }
}
```

Nested interface inside another interface Example

```
public class NestedInterfaceExample {

        public static void main(String[] args) {
                OnlyOuter obj_1 = new OnlyOuter();
                OnlyInner obj_2 = new OnlyInner();

                obj_1.outerMethod();
                obj_2.innerMethod();
        }

}
```

Nested interface inside a class Example

- The nested interface that defined inside a class must be accessed as `ClassName.InnerInterface`

```
class OuterClass{

        interface InnerInterface{
                void innerMethod();
        }
}
```

Nested interface inside a class Example

```
class ImplementingClass implements OuterClass.InnerInterface{
        public void innerMethod() {
                System.out.println("This is InnerInterface method");
        }
}
```

Nested interface inside a class Example

```
public class NestedInterfaceExample {

        public static void main(String[] args) {
                ImplementingClass obj = new ImplementingClass();

                obj.innerMethod();
        }

}
```

Variables in Java Interfaces

Variables in Java Interfaces

- In java, an interface is a completely abstract class.
- An interface is a container of abstract methods and static final variables.
- The interface contains the static final variables.
- The variables defined in an interface can not be modified by the class that implements the interface,
    - but it may use as it defined in the interface.

Variables in Java Interfaces

- The variable in an interface is public, static, and final by default.
- If any variable in an interface is defined without public, static, and final keywords then, the compiler automatically adds the same.
- No access modifier is allowed except the public for interface variables.
- Every variable of an interface must be initialized in the interface itself.
- The class that implements an interface can not modify the interface variable, but it may use as it defined in the interface.

Variables in Java Interfaces Example-1

```
interface SampleInterface{

        int UPPER_LIMIT = 100;

        //int LOWER_LIMIT; // Error - must be initialised

}
```

```
public class InterfaceVariablesExample implements SampleInterface{

        public static void main(String[] args) {

                System.out.println("UPPER LIMIT = " + UPPER_LIMIT);

                // UPPER_LIMIT = 150; // Can not be modified
        }
}
```

Extending an Interface in java

Extending an Interface in java

- In java, an interface can extend another interface.
  - When an interface wants to extend another interface,
    - it uses the keyword extends.
- The interface that extends another interface has its own members and all the members defined in its parent interface too.
- The class which implements a child interface needs to provide code for the methods defined in both child and parent interfaces,
  - otherwise, it needs to be defined as abstract class.

Extending an Interface in Java

- An interface can extend another interface.
- An interface can not extend multiple interfaces.
- An interface can implement neither an interface nor a class.
- The class that implements child interface needs to provide code for all the methods defined in both child and parent interfaces.

Extending an Interface in Java Example-1

```
interface ParentInterface{
        void parentMethod();
}
```

```
interface ChildInterface extends ParentInterface{
        void childMethod();
}
```

Extending an Interface in Java Example-1

```
class ImplementingClass implements ChildInterface{

        public void childMethod() {
                System.out.println("Child Interface method!!");
        }

        public void parentMethod() {
                System.out.println("Parent Interface mehtod!");
        }
}
```

Extending an Interface in Java Example-1

```
public class ExtendingAnInterface {

        public static void main(String[] args) {

                ImplementingClass obj = new ImplementingClass();

                obj.childMethod();
                obj.parentMethod();

        }

}
```

Extending an Interface in Java Example-2

- Here, the Polygon interface extends the Line interface. Now, if any class implements Polygon, it should provide implementations for all the abstract methods of both Line and Polygon

```
interface Line {
  // members of Line interface
}

// extending interface
interface Polygon extends Line {
  // members of Polygon interface
  // members of Line interface
}
```

Extending Multiple Interfaces in Java Example

```
interface A {
    ...
}
```

```
interface B {
    ...
}
```

```
interface C extends A, B {
    ...
}
```

Advantages of Interface in Java

Advantages of Interface in Java

- **Similar to abstract classes, interfaces help us to achieve abstraction in Java**
  - Here, we know getArea() calculates the area of polygons but the way area is calculated is different for different polygons. Hence, the implementation of getArea() is independent of one another.

Advantages of Interface in Java

- **Interfaces provide specifications that a class (which implements it) must follow.**
  - In our previous example, we have used getArea() as a specification inside the interface Polygon. This is like setting a rule that we should be able to get the area of every polygon.
- Now any class that implements the Polygon interface must provide an implementation for the getArea() method.

Advantages of Interface in Java

- Interfaces are also used to achieve multiple inheritance in Java
  - In the example, the class Rectangle is implementing two different interfaces. This is how we achieve multiple inheritance in Java.

```
interface Line {
…
}

interface Polygon {
…
}

class Rectangle implements Line, Polygon {
…
}
```

Advantages of Interface in Java

- All the methods inside an interface are implicitly public and all fields are implicitly public static final. For example,

```
interface Language {

  // by default public static final
  String type = "programming language";

  // by default public
  void getName();
}
```

default methods in Java Interfaces

default methods in Java Interfaces

- With the release of Java 8, we can now add methods with implementation inside an interface.
  - These methods are called default methods.
- To declare default methods inside interfaces, we use the default keyword. For example,

```
public default void getSides() {
   // body of getSides()
}
```

why default methods in Java Interfaces

- Let's take a scenario to understand why default methods are introduced in Java.
- Suppose, we need to add a new method in an interface.
- We can add the method in our interface easily without implementation. However, that's not the end of the story. All our classes that implement that interface must provide an implementation for the method.
- If a large number of classes were implementing this interface, we need to track all these classes and make changes to them. This is not only tedious but error-prone as well.
- To resolve this, Java introduced default methods. Default methods are inherited like ordinary methods.

Default Method in Java Interface Example

```
interface Polygon {
  void getArea();

  // default method
  default void getSides() {
    System.out.println("I can get sides of a polygon.");
  }
}
```

Default Method in Java Interface Example

```
// implements the interface
class Rectangle implements Polygon {
  public void getArea() {
    int length = 6;
    int breadth = 5;
    int area = length * breadth;
    System.out.println("The area of the rectangle is " + area);
  }

  // overrides the getSides()
  public void getSides() {
    System.out.println("I have 4 sides.");
  }
}
```

Default Method in Java Interface Example

```
// implements the interface
class Square implements Polygon {
  public void getArea() {
    int length = 5;
    int area = length * length;
    System.out.println("The area of the square is " + area);
  }
}
```

Default Method in Java Interface Example

```
class Main {
  public static void main(String[] args) {

    // create an object of Rectangle
    Rectangle r1 = new Rectangle();
    r1.getArea();
    r1.getSides();

    // create an object of Square
    Square s1 = new Square();
    s1.getArea();
    s1.getSides();
  }
}
```

Default Method in Java Interface Example

- In the example, we have created an interface named Polygon. It has a default method getSides() and an abstract method getArea().
- Here, we have created two classes Rectangle and Square that implement Polygon.
- The Rectangle class provides the implementation of the getArea() method and overrides the getSides() method. However, the Square class only provides the implementation of the getArea() method.
- Now, while calling the getSides() method using the Rectangle object, the overridden method is called. However, in the case of the Square object, the default method is called.

private and static Methods in Interface

private and static Methods in Interface

- The Java 8 also added another feature to include static methods inside an interface.
- Similar to a class, we can access static methods of an interface using its references. For example,

```
// create an interface
interface Polygon {
  staticMethod(){..}
}

// access static method
Polygon.staticMethod();
```

private and static Methods in Interface

- Note: With the release of Java 9, private methods are also supported in interfaces.
- We cannot create objects of an interface.
  - Hence, private methods are used as helper methods that provide support to other methods in interfaces.

Java Interface Practical Example

```
// To use the sqrt function
import java.lang.Math;

interface  Polygon {
   void getArea();

 // calculate the perimeter of a Polygon
   default void getPerimeter(int... sides) {
      int perimeter = 0;
      for (int side: sides) {
         perimeter += side;
      }

   System.out.println("Perimeter: " + perimeter);
   }
}
```

Java Interface Practical Example

```
class Triangle implements Polygon {
   private int a, b, c;
   private double s, area;

// initializing sides of a triangle
   Triangle(int a, int b, int c) {
      this.a = a;
      this.b = b;
      this.c = c;
      s = 0;
   }

// calculate the area of a triangle
   public void getArea() {
      s = (double) (a + b + c)/2;
      area = Math.sqrt(s*(s-a)*(s-b)*(s-c));
      System.out.println("Area: " + area);
   }
}
```

Java Interface Practical Example

```
class Main {
   public static void main(String[] args) {
      Triangle t1 = new Triangle(2, 3, 4);

// calls the method of the Triangle class
      t1.getArea();

// calls the method of Polygon
      t1.getPerimeter(2, 3, 4);
   }
}
```

Java Interface Practical Example

- In the example, we have created an interface named Polygon.
  - It includes a default method getPerimeter() and an abstract method getArea().
- We can calculate the perimeter of all polygons in the same manner so we implemented the body of getPerimeter() in Polygon.
- Now, all polygons that implement Polygon can use getPerimeter() to calculate perimeter.
- However, the rule for calculating the area is different for different polygons.
  - Hence, getArea() is included without implementation.
- Any class that implements Polygon must provide an implementation of getArea().

Java Reflection

Java Reflection

- In Java, reflection allows us to inspect and manipulate classes, interfaces, constructors, methods, and fields at run time.
- There is a class in Java named `Class` that keeps all the information about objects and classes at runtime. The object of `Class` can be used to perform reflection.

Reflection of Java Classes

- In order to reflect a Java class, we first need to create an object of Class.
- And, using the object we can call various methods to get information about methods, fields, and constructors present in a class.
- There exists three ways to create objects of Class:

Reflection of Java Classes

- Using forName() method

```
class Dog {...}

// create object of Class
// to reflect the Dog class
Class a = Class.forName("Dog");
```

- Here, the forName() method takes the name of the class to be reflected as its argument.

Reflection of Java Classes

- Using getClass() method

```
// create an object of Dog class
Dog d1 = new Dog();

// create an object of Class
// to reflect Dog
Class b = d1.getClass();
```

- Here, we are using the object of the Dog class to create an object of Class.

Reflection of Java Classes

- Using .class extension

```
// create an object of Class
// to reflect the Dog class
Class c = Dog.class;
```

- Now that we know how we can create objects of the Class.
  - We can use this object to get information about the corresponding class at runtime.

Java Class Reflection Example

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

class Animal {
}

// put this class in different Dog.java file
public class Dog extends Animal {
  public void display() {
    System.out.println("I am a dog.");
  }
}
```

Java Class Reflection Example

```
// put this in Main.java file
class Main {
  public static void main(String[] args) {
    try {
      // create an object of Dog
      Dog d1 = new Dog();
      // create an object of Class
      // using getClass()
      Class obj = d1.getClass();
      // get name of the class
      String name = obj.getName();
      System.out.println("Name: " + name);
      // get the access modifier of the class
      int modifier = obj.getModifiers();
      // convert the access modifier to string
      String mod = Modifier.toString(modifier);
      System.out.println("Modifier: " + mod);
      // get the superclass of Dog
      Class superClass = obj.getSuperclass();
      System.out.println("Superclass: " + superClass.getName());
    }catch (Exception e) { e.printStackTrace();}
  }
}
```

Java Class Reflection Example

- In the example, we have created a superclass: Animal and a subclass: Dog. Here, we are trying to inspect the class Dog.
- Notice the statement,

```
Class obj = d1.getClass();
```

- Here, we are creating an object obj of Class using the getClass() method. Using the object, we are calling different methods of Class.
  - **obj.getName()** - returns the name of the class
  - **obj.getModifiers()** - returns the access modifier of the class
  - **obj.getSuperclass()** - returns the super class of the class
- **Note:** We are using the Modifier class to convert the integer access modifier to a string.

Reflecting Fields, Methods, and Constructors

- The package java.lang.reflect provides classes that can be used for manipulating class members. For example,
- **Method class** - provides information about methods in a class
- **Field class** - provides information about fields in a class
- **Constructor class** - provides information about constructors in a class

Reflection of Java Methods

- The Method class provides various methods that can be used to get information about the methods present in a class.

Reflection of Java Methods Example

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

class Dog {

  // methods of the class
  public void display() {
    System.out.println("I am a dog.");
  }

  private void makeSound() {
    System.out.println("Bark Bark");
  }
}
```

Reflection of Java Methods Example

```
class Main {
  public static void main(String[] args) {
    try {

      // create an object of Dog
      Dog d1 = new Dog();

      // create an object of Class
      // using getClass()
      Class obj = d1.getClass();

      // using object of Class to
      // get all the declared methods of Dog
      Method[] methods = obj.getDeclaredMethods();
...
```

Reflection of Java Methods Example

```
...
      // create an object of the Method class
      for (Method m : methods) {

        // get names of methods
        System.out.println("Method Name: " + m.getName());

        // get the access modifier of methods
        int modifier = m.getModifiers();
        System.out.println("Modifier: " + Modifier.toString(modifier));

        // get the return types of method
        System.out.println("Return Types: " + m.getReturnType());
        System.out.println(" ");
      }
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Reflection of Java Methods Example

- In the example, we are trying to get information about the methods present in the Dog class.
- As mentioned earlier, we have first created an object obj of Class using the getClass() method.
- Notice the expression,

```
Method[] methods = obj.getDeclaredMethod();
```

- Here, the getDeclaredMethod() returns all the methods present inside the class.

Reflection of Java Methods Example

- Also, we have created an object m of the Method class. Here,
  - `m.getName()` - returns the name of a method
  - `m.getModifiers()` - returns the access modifier of methods in integer form
  - `m.getReturnType()` - returns the return type of methods
- The Method class also provides various other methods that can be used to inspect methods at run time.

Reflection of Java Fields

- Like methods, we can also inspect and modify different fields of a class using the methods of the Field class.

Reflection of Java Public Fields Example

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

class Dog {
  public String type;
}
```

Reflection of Java Public Fields Example

```
class Main {
  public static void main(String[] args) {
    try {
      // create an object of Dog
      Dog d1 = new Dog();

      // create an object of Class
      // using getClass()
      Class obj = d1.getClass();

      // access and set the type field
      Field field1 = obj.getField("type");
      field1.set(d1, "labrador");
...
```

Reflection of Java Public Fields Example

```
...
      // get the value of the field type
      String typeValue = (String) field1.get(d1);
      System.out.println("Value: " + typeValue);

      // get the access modifier of the field type
      int mod = field1.getModifiers();

      // convert the modifier to String form
      String modifier1 = Modifier.toString(mod);
      System.out.println("Modifier: " + modifier1);
      System.out.println(" ");
    }

    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Reflection of Java Public Fields Example

- In the example, we have created a class named Dog.
  - It includes a public field named type. Notice the statement,

```
Field field1 = obj.getField("type");
```

- Here, we are accessing the public field of the Dog class and assigning it to the object field1 of the Field class.
- We then used various methods of the Field class:
  - `field1.set()` - sets the value of the field
  - `field1.get()` - returns the value of field
  - `field1.getModifiers()` - returns the value of the field in integer form

Reflection of Java Private Fields Example

- Similarly, we can also access and modify private fields as well. However, the reflection of private field is little bit different than the public field

Reflection of Java Private Fields Example

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

class Dog {
  private String color;
}
```

Reflection of Java Private Fields Example

```
class Main {
  public static void main(String[] args) {
    try {
      // create an object of Dog
      Dog d1 = new Dog();

      // create an object of Class
      // using getClass()
      Class obj = d1.getClass();

      // access the private field color
      Field field1 = obj.getDeclaredField("color");

      // allow modification of the private field
      field1.setAccessible(true);
...
```

Reflection of Java Private Fields Example

```
...
      // set the value of color
      field1.set(d1, "brown");

      // get the value of field color
      String colorValue = (String) field1.get(d1);
      System.out.println("Value: " + colorValue);

      // get the access modifier of color
      int mod2 = field1.getModifiers();

      // convert the access modifier to string
      String modifier2 = Modifier.toString(mod2);
      System.out.println("Modifier: " + modifier2);
    }

    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Reflection of Java Private Fields Example

- In the example, we have created a class named `Dog`.
- The class contains a private field named `color`. Notice the statement.

```
Field field1 = obj.getDeclaredField("color");
field1.setAccessible(true);
```

- Here, we are accessing color and assigning it to the object `field1` of the Field class.
- We then used `field1` to modify the accessibility of color and allows us to make changes to it.
- We then used `field1` to perform various operations on the private field color.

Reflection of Java Constructor

- We can also inspect different constructors of a class using various methods provided by the Constructor class

Reflection of Java Constructor Example

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

class Dog {

  // public constructor without parameter
  public Dog() {

  }

  // private constructor with a single parameter
  private Dog(int age) {

  }

}
```

Reflection of Java Constructor Example

```
class Main {
  public static void main(String[] args) {
    try {
      // create an object of Dog
      Dog d1 = new Dog();

      // create an object of Class
      // using getClass()
      Class obj = d1.getClass();

      // get all constructors of Dog
      Constructor[] constructors = obj.getDeclaredConstructors();
...
```

Reflection of Java Constructor Example

```
...
      for (Constructor c : constructors) {

        // get the name of constructors
        System.out.println("Constructor Name: " + c.getName());

        // get the access modifier of constructors
        // convert it into string form
        int modifier = c.getModifiers();
        String mod = Modifier.toString(modifier);
        System.out.println("Modifier: " + mod);

        // get the number of parameters in constructors
        System.out.println("Parameters: " + c.getParameterCount());
        System.out.println("");
      }
    }

    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Reflection of Java Constructor Example

- In the example, we have created a class named Dog. The class includes two constructors.
- We are using reflection to find the information about the constructors of the class. Notice the statement,

```
Constructor[] constructors = obj.getDeclaredConstructor();
```

- Here, the we are accessing all the constructors present in Dog and assigning them to an array constructors of the Constructor type.
- We then used object c to get different informations about the constructor.
  - `c.getName()` - returns the name of the constructor
  - `c.getModifiers()` - returns the access modifiers of the constructor in integer form
  - `c.getParameterCount()` - returns the number of parameters present in each constructor

Java Wrapper Classes

Java Wrapper Classes

- Wrapper classes provide a way to use primitive data types (int, boolean, etc..) as objects

Primitive Data Type $\Longrightarrow$ Wrapper Class

byte $\Longrightarrow$ Byte

short $\Longrightarrow$ Short

int $\Longrightarrow$ Integer

long $\Longrightarrow$ Long

float $\Longrightarrow$ Float

double $\Longrightarrow$ Double

boolean $\Longrightarrow$ Boolean

char $\Longrightarrow$ Character

Java Wrapper Classes

- Sometimes you must use wrapper classes, for example when working with Collection objects, such as ArrayList, where primitive types cannot be used (the list can only store objects)

```
ArrayList<int> myNumbers = new ArrayList<int>(); // Invalid
```

```
ArrayList<Integer> myNumbers = new ArrayList<Integer>(); // Valid
```

Creating Wrapper Objects

- To create a wrapper object, use the wrapper class instead of the primitive type. To get the value, you can just print the object

```
public class Main {
  public static void main(String[] args) {
    Integer myInt = 5;
    Double myDouble = 5.99;
    Character myChar = 'A';
    System.out.println(myInt);
    System.out.println(myDouble);
    System.out.println(myChar);
  }
}
```

Creating Wrapper Objects

- Since you're now working with objects, you can use certain methods to get information about the specific object.
- For example, the following methods are used to get the value associated with the corresponding wrapper object: `intValue()`, `byteValue()`, `shortValue()`, `longValue()`, `floatValue()`, `doubleValue()`, `charValue()`, `booleanValue()`.

Creating Wrapper Objects

```
public class Main {
  public static void main(String[] args) {
    Integer myInt = 5;
    Double myDouble = 5.99;
    Character myChar = 'A';
    System.out.println(myInt.intValue());
    System.out.println(myDouble.doubleValue());
    System.out.println(myChar.charValue());
  }
}
```

Creating Wrapper Objects

- Another useful method is the toString() method, which is used to convert wrapper objects to strings.
- In the following example, we convert an Integer to a String, and use the length() method of the String class to output the length of the "string":

```
public class Main {
  public static void main(String[] args) {
    Integer myInt = 100;
    String myString = myInt.toString();
    System.out.println(myString.length());
  }
}
```

Java Lambda Expressions

Java Lambda Expressions

- Lambda Expressions were added in Java 8.
- A lambda expression is a short block of code which takes in parameters and returns a value. Lambda expressions are similar to methods, but they do not need a name and they can be implemented right in the body of a method.

Java Lambda Expressions Syntax

- The simplest lambda expression contains a single parameter and an expression:

```
parameter -> expression
```

- To use more than one parameter, wrap them in parentheses:

```
(parameter1, parameter2) -> expression
```

- Expressions are limited. They have to immediately return a value, and they cannot contain variables, assignments or statements such as if or for. In order to do more complex operations, a code block can be used with curly braces. If the lambda expression needs to return a value, then the code block should have a return statement.

```
(parameter1, parameter2) -> { code block }
```

Lambda expressions are usually passed as parameters to a function

Using Lambda Expressions

- Use a lamba expression in the ArrayList's forEach() method to print every item in the list

```
import java.util.ArrayList;

public class Main {
  public static void main(String[] args) {
    ArrayList<Integer> numbers = new ArrayList<Integer>();
    numbers.add(5);
    numbers.add(9);
    numbers.add(8);
    numbers.add(1);
    numbers.forEach( (n) -> { System.out.println(n); } );
  }
}
```

Lambda expressions can be stored in variables if the variable's type is an interface which has only one method. The lambda expression should have the same number of parameters and the same return type as that method. Java has many of these kinds of interfaces built in, such as the Consumer interface (found in the java.util package) used by lists.

Using Lambda Expressions

- Use Java's Consumer interface to store a lambda expression in a variable:

```
import java.util.ArrayList;
import java.util.function.Consumer;

public class Main {
  public static void main(String[] args) {
    ArrayList<Integer> numbers = new ArrayList<Integer>();
    numbers.add(5);
    numbers.add(9);
    numbers.add(8);
    numbers.add(1);
    Consumer<Integer> method = (n) -> { System.out.println(n); };
    numbers.forEach( method );
  }
}
```

To use a lambda expression in a method, the method should have a parameter with a single-method interface as its type. Calling the interface's method will run the lambda expression:

Using Lambda Expressions

- Create a method which takes a lambda expression as a parameter:

```
interface StringFunction {
  String run(String str);
}

public class Main {
  public static void main(String[] args) {
    StringFunction exclaim = (s) -> s + "!";
    StringFunction ask = (s) -> s + "?";
    printFormatted("Hello", exclaim);
    printFormatted("Hello", ask);
  }
  public static void printFormatted(String str, StringFunction format) {
    String result = format.run(str);
    System.out.println(result);
  }
}
```

References

- BTechSmartClass-Defining an Interface in Java
- BTechSmartClass-Implementing an Interface in Java
- BTechSmartClass-Nested Interfaces in java
- BTechSmartClass-Variables in Java Interfaces
- BTechSmartClass-Extending an Interface in java
- Programiz-Java Interface
- Programiz-Java Reflection
- W3schools-Java Wrapper Classes
- W3schools-Java Lambda Expressions