

CE204 Object-Oriented Programming

OOP with Java-III

Author: Asst. Prof. Dr. Uğur CORUH

Contents

0.1	CE204 Object-Oriented Programming	2
0.1.1	Week-3 (OOP with Java-III)	2
0.2	OOP with Java-III	2
0.2.1	Outline (1)	2
0.2.2	Outline (2)	2
0.3	Defining an Interface in Java	3
0.3.1	Defining an Interface in Java	3
0.3.2	Defining an Interface in Java	3
0.3.3	Defining an Interface in Java	3
0.3.4	Defining an Interface in Java	3
0.4	Implementing an Interface in Java	4
0.4.1	Implementing an Interface in Java	4
0.4.2	Implementing an Interface in Java	4
0.4.3	Implementing an Interface in Java Example-1	4
0.4.4	Implementing an Interface in Java Example-1	5
0.4.5	Implementing an Interface in Java Example-3	6
0.5	Implementing multiple Interfaces	6
0.6	Implementing multiple Interfaces	6
0.7	Implementing multiple Interfaces Example-1	6
0.8	Implementing multiple Interfaces Example-1	6
0.9	Implementing multiple Interfaces Example-1	7
0.10	Implementing multiple Interfaces Example-2	7
0.11	Nested Interfaces in Java	7
0.11.1	Nested Interfaces in Java	7
0.11.2	Nested Interfaces in Java	7
0.11.3	Nested Interfaces in Java	7
0.11.4	Nested interface inside another interface Example	7
0.11.5	Nested interface inside another interface Example	8
0.12	Nested interface inside a class Example	8
0.13	Nested interface inside a class Example	8
0.13.1	Extending an Interface in Java Example-1	10
0.13.2	Extending an Interface in Java Example-2	11
0.14	Advantages of Interface in Java	11
0.14.1	Advantages of Interface in Java	11
0.14.2	Advantages of Interface in Java	11
0.14.3	Advantages of Interface in Java	11
0.15	default methods in Java Interfaces	12
0.15.1	default methods in Java Interfaces	12
0.15.2	why default methods in Java Interfaces	12
0.15.3	Default Method in Java Interface Example	12
0.15.4	Default Method in Java Interface Example	13
0.15.5	Default Method in Java Interface Example	13
0.15.6	private and static Methods in Interface	14

0.15.7	Java Interface Practical Example	14
0.15.8	Java Interface Practical Example	15
0.15.9	Reflection of Java Classes	16
0.15.10	Java Class Reflection Example	16
0.15.11	Java Class Reflection Example	17
0.15.12	Reflecting Fields, Methods, and Constructors	17
0.16	Java Wrapper Classes	22
0.16.1	Java Wrapper Classes	22
0.16.2	Java Wrapper Classes	23
0.16.3	Creating Wrapper Objects	23
0.16.4	Creating Wrapper Objects	23
0.16.5	Creating Wrapper Objects	23
0.16.6	Creating Wrapper Objects	23
0.17	Java Lambda Expressions	23
0.17.1	Java Lambda Expressions	23
0.17.2	Java Lambda Expressions Syntax	24
0.17.3	Java Lambda Expressions Syntax	24
0.17.4	Using Lambda Expressions	24
0.17.5	Using Lambda Expressions	25
0.18	References	25

List of Figures

List of Tables

0.1 CE204 Object-Oriented Programming

0.1.1 Week-3 (OOP with Java-III)

0.1.1.1 Spring Semester, 2021-2022 Download DOC-PDF¹, DOC-DOCX², SLIDE³, PPTX⁴,

0.2 OOP with Java-III

0.2.1 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
-

0.2.2 Outline (2)

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

¹ce204-week-3.en.md_doc.pdf

²ce204-week-3.en.md_word.docx

³ce204-week-3.en.md_slide.pdf

⁴ce204-week-3.en.md_slide.pptx

- Java Lambda Expressions
-

0.3 Defining an Interface in Java

0.3.1 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.
-

0.3.2 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.
-

0.3.3 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.

```
Java linenums="1" interface InterfaceName{    ...    members declaration;    ... }
```

0.3.4 Defining an Interface in Java

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

```
““ Java linenums="1" 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.

```
``` Java linenums="1"
interface HumanInterfaceExample {

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

 public static final int duration = 10;

}
```

---

## 0.4 Implementing an Interface in Java

---

### 0.4.1 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.

---

### 0.4.2 Implementing an Interface in Java

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

```
Java linenums="1" class className implements InterfaceName{ ... boby-of-the-class
... }
```

---

### 0.4.3 Implementing an Interface in Java Example-1

```
““ Java linenums="1" interface Human {

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

int duration = 10;

}
```

```

```

### ### Implementing an Interface in Java Example-1

```
``` Java linenums="1"
class Programmer implements Human{
    public void learn(String str) {
        System.out.println("Learn using " + str);
    }
    public void work() {
        System.out.println("Develop applications");
    }
}
}
```

0.4.4 Implementing an Interface in Java Example-1

```
``` Java linenums="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

---

### ### Implementing an Interface in Java Example-2

```
``` Java linenums="1"
interface Polygon {
    void getArea(int length, int breadth);
}
}
```

```
``` Java linenums="1" // 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

```
``` Java linenums="1"
class Main {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle();
        r1.getArea(5, 6);
    }
}
```

```
}
```

0.4.5 Implementing an Interface in Java Example-3

```
Java lineums="1" // create an interface interface Language { void getName(String
name); }
““ Java lineums="1" // 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

```
``` Java lineums="1"
class Main {
 public static void main(String[] args) {
 ProgrammingLanguage language = new ProgrammingLanguage();
 language.getName("Java");
 }
}
```

---

## 0.5 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.

---

## 0.6 Implementing multiple Interfaces

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

```
Java lineums="1" class className implements InterfaceName1, InterfaceName2, ...{ ...
body-of-the-class ... }
```

---

## 0.7 Implementing multiple Interfaces Example-1

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

```
Java lineums="1" interface Human { void learn(String str); void work(); }
Java lineums="1" interface Recruitment { boolean screening(int score); boolean
interview(boolean selected); } —
```

---

## 0.8 Implementing multiple Interfaces Example-1

```
Java lineums="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"); } }
```

---

## 0.9 Implementing multiple Interfaces Example-1

```
Java lineums="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.
- 

## 0.10 Implementing multiple Interfaces Example-2

```
Java lineums="1" interface A { // members of A }
Java lineums="1" interface B { // members of B }
Java lineums="1" class C implements A, B { // abstract members of A // abstract
members of B }
```

---

## 0.11 Nested Interfaces in Java

---

### 0.11.1 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 known as nested interface.
    - The nested interface is also referred as inner interface.
- 

### 0.11.2 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.
- 

### 0.11.3 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.
- 

### 0.11.4 Nested interface inside another interface Example

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

```
““ Java lineums="1" interface OuterInterface{ void outerMethod();
interface InnerInterface{
 void innerMethod();
}
```

```
}
```

```

```

```
Nested interface inside another interface Example
```

```
``` Java linenums="1"
```

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

```
Java linenums="1" class OnlyInner implements OuterInterface.InnerInterface{    public  
void innerMethod() {        System.out.println("This is InnerInterface method");    } }
```

0.11.5 Nested interface inside another interface Example

```
``` Java linenums="1" 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``

```
``` Java linenums="1"
```

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

0.12 Nested interface inside a class Example

```
Java linenums="1" class ImplementingClass implements OuterClass.InnerInterface{    public  
void innerMethod() {        System.out.println("This is InnerInterface method");    } }
```

0.13 Nested interface inside a class Example

```
``` Java linenums="1" 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 will give an error.
- 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

```

``` Java linenums="1"
interface SampleInterface{

    int UPPER_LIMIT = 100;

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

``` Java linenums="1" 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.
- The class which implements a child interface needs to provide code for the methods defined in both child and parent interface.
  - 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 interface.

---

### ### Extending an Interface in Java Example-1

```
``` Java linenums="1"
interface ParentInterface{
    void parentMethod();
}

Java linenums="1" interface ChildInterface extends ParentInterface{
    void childMethod();
}
```

0.13.1 Extending an Interface in Java Example-1

```
``` Java linenums="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

```
``` Java linenums="1"
public class ExtendingAnInterface {

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

```

        ImplementingClass obj = new ImplementingClass();

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

```

0.13.2 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

```

““ Java linenums="1" 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

```

``` Java linenums="1"
interface A {
 ...
}

Java linenums="1" interface B { ... }

Java linenums="1" interface C extends A, B { ... }

```

---

## 0.14 Advantages of Interface in Java

---

### 0.14.1 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.
- 

### 0.14.2 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.
- 

### 0.14.3 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.

```
““ Java linenums=“1” 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

```
``` Java linenums="1"
interface Language {

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

    // by default public
    void getName();
}

```

0.15 default methods in Java Interfaces

0.15.1 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,

```
Java linenums="1" public default void getSides() { // body of getSides() }
```

0.15.2 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.

0.15.3 Default Method in Java Interface Example

```
““ Java linenums=“1” 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

```

` `` Java linenums="1"
// 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.");
    }
}

```

0.15.4 Default Method in Java Interface Example

```

Java linenums="1" // 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); } }

```

0.15.5 Default Method in Java Interface Example

```

“ Java linenums="1" 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
- 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() m
- Now, while calling the getSides() method using the Rectangle object, the overridden 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,

```

``` Java linenums="1"
// create an interface
interface Polygon {
 staticMethod(){..}
}

// access static method
Polygon.staticMethod();

```

---

### 0.15.6 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.
- 

### 0.15.7 Java Interface Practical Example

```

``` Java linenums="1" // 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

```

``` Java linenums="1"
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);
 }
}

```

---

### 0.15.8 Java Interface Practical Example

```
““ Java linenums="1" 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()`.
- 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 attributes.
- There is a class in Java named `Class` that keeps all the information about objects and classes at runtime.

---

#### ### 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.
- There exists three ways to create objects of `Class`:

---

#### ### Reflection of Java Classes

- Using `forName()` method

```
``` Java linenums="1"
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.

0.15.9 Reflection of Java Classes

- Using getClass() method

```
““ Java linenums=“1” // 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

```
``` Java linenums="1"
// 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.

---

### 0.15.10 Java Class Reflection Example

```
““ Java linenums=“1” 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() { Sys-
tem.out.println("I am a dog."); } }
```

---

```
Java Class Reflection Example
```

```
``` Java linenums="1"
// 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(); }
    }
}
```



```
}
```

0.15.11 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,

```
Java linenums="1" 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.
-

0.15.12 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
-

0.15.12.1 Reflection of Java Methods

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

0.15.12.2 Reflection of Java Methods Example

```
““ Java linenums="1" 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

```
``` Java linenums="1"  
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();
...

```

---

**0.15.12.3 Reflection of Java Methods Example** “Java linenums=“1” ... // 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,

```

``` Java linenums="1"
Method[] methods = obj.getDeclaredMethod();

```

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

0.15.12.4 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.
-

0.15.12.5 Reflection of Java Fields

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

0.15.12.6 Reflection of Java Public Fields Example ““ Java linenums=“1” import java.lang.Class; import java.lang.reflect.*; class Dog { public String type; } ---

Reflection of Java Public Fields Example

```

``` Java linenums="1"
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");
 }
 ...
 }
}

```

**0.15.12.7 Reflection of Java Public Fields Example** ““ Java linenums=“1” ... // 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,

```

``` Java linenums="1"
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

0.15.12.8 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
-

0.15.12.9 Reflection of Java Private Fields Example “ Java linenums=“1” import java.lang.Class; import java.lang.reflect.*;

```
class Dog { private String color; }
```

```
---
```

Reflection of Java Private Fields Example

```
``` Java linenums="1"
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);
 }
 ...
 }
}
```

---

#### 0.15.12.10 Reflection of Java Private Fields Example “ Java linenums=“1” ... // 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.

```
``` Java linenums="1"
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.

0.15.12.11 Reflection of Java Constructor

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

0.15.12.12 Reflection of Java Constructor Example

```
““ Java linenums="1" 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

```
``` Java linenums="1"
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();
 }
 ...
 }
}
```

---

#### 0.15.12.13 Reflection of Java Constructor Example

```
““ Java linenums="1" ... 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 state

```
``` Java linenums="1"
```

```
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.

0.15.12.14 Reflection of Java Constructor Example

- 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

0.16 Java Wrapper Classes

0.16.1 Java Wrapper Classes

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

Primitive Data Type \impl Wrapper Class
 byte \impl Byte short \impl Short int \impl Integer long \impl Long float \impl Float double \impl Double boolean \impl Boolean char \impl Character

0.16.2 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)

```
Java linenums="1" ArrayList<int> myNumbers = new ArrayList<int>(); // Invalid
```

```
Java linenums="1" ArrayList<Integer> myNumbers = new ArrayList<Integer>(); // Valid
```

0.16.3 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

```
Java linenums="1" 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); } }
```

0.16.4 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()`.
-

0.16.5 Creating Wrapper Objects

```
Java linenums="1" 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());
} }
```

0.16.6 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":

```
Java linenums="1" public class Main { public static void main(String[] args) { Integer
myInt = 100; String myString = myInt.toString(); System.out.println(myString.length());
} }
```

0.17 Java Lambda Expressions

0.17.1 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.

0.17.2 Java Lambda Expressions Syntax

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

```
Java linenums="1" parameter -> expression
```

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

```
Java linenums="1" (parameter1, parameter2) -> expression
```

0.17.3 Java Lambda Expressions Syntax

- 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.

```
Java linenums="1" (parameter1, parameter2) -> { code block }
```

Lambda expressions are usually passed as parameters to a function

0.17.4 Using Lambda Expressions

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

```
““ Java linenums="1" import java.util.ArrayList;
```

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

```
---
```

- **Lambda expressions can be stored in variables if the variable's type is an interface which has only
- **The lambda expression should have the same number of parameters and the same return type as that me
- **Java has many of these kinds of interfaces built in, such as the Consumer interface (found in the j

```
---
```

```
### Using Lambda Expressions
```

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

```
``` Java linenums="1"  
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:
- 

### 0.17.5 Using Lambda Expressions

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

```
““ Java linenums=“1” interface StringFunction { String run(String str); }
```

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

---

## 0.18 References

- BTechSmartClass-Defining an Interface in Java<sup>5</sup>
  - BTechSmartClass-Implementing an Interface in Java<sup>6</sup>
  - BTechSmartClass-Nested Interfaces in java<sup>7</sup>
  - BTechSmartClass-Variables in Java Interfaces<sup>8</sup>
  - BTechSmartClass-Extending an Interface in java<sup>9</sup>
  - Programiz-Java Interface<sup>10</sup>
  - Programiz-Java Reflection<sup>11</sup>
  - W3schools-Java Wrapper Classes<sup>12</sup>
  - W3schools-Java Lambda Expressions<sup>13</sup>
- 

*End – Of – Week – 3 – Module*

---

<sup>5</sup><http://www.btechsmartclass.com/java/java-defining-an-interface.html>

<sup>6</sup><http://www.btechsmartclass.com/java/java-implementing-an-interface.html>

<sup>7</sup><http://www.btechsmartclass.com/java/java-nested-interfaces.html>

<sup>8</sup><http://www.btechsmartclass.com/java/java-variables-in-interfaces.html>

<sup>9</sup><http://www.btechsmartclass.com/java/java-extending-an-interface.html>

<sup>10</sup><https://www.programiz.com/java-programming/interfaces>

<sup>11</sup><https://www.programiz.com/java-programming/reflection>

<sup>12</sup>[https://www.w3schools.com/java/java\\_wrapper\\_classes.asp](https://www.w3schools.com/java/java_wrapper_classes.asp)

<sup>13</sup>[https://www.w3schools.com/java/java\\_lambda.asp](https://www.w3schools.com/java/java_lambda.asp)