Java Programming
Chapter 8

- Inner Class
- Abtract
- Interface

Java Inner Classes

- Java Inner Classes
  - In Java, it is also possible to nest classes (a class within a class).
  - The purpose of nested classes is to group classes that belong together, which makes your code more readable and maintainable.
  - To access the inner class, create an object of the outer class, and then create an object of the inner class:

Java Inner Classes

```java
class OuterClass {
  int x = 10;

  class InnerClass {
    int y = 5;
  }
}

public class Main {
  public static void main(String[] args) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    System.out.println(myInner.y + myOuter.x);
  }
}
```

Private Inner Class

- Unlike a "regular" class, an inner class can be private or protected. If you don't want outside objects to access the inner class, declare the class as private:
- If you try to access a private inner class from an outside class, an error occurs:

```
class OuterClass {
  int x = 10;

  private class InnerClass {
    int y = 5;
  }
}

public class Main {
  public static void main(String[] args) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    System.out.println(myInner.y + myOuter.x);
  }
}
```

Abstract Classes and Methods

- Data **abstraction** is the process of hiding certain details and showing only essential information to the user.
- Abstraction can be achieved with either abstract classes or interfaces -> (which you will learn more about in the next chapter).
- The **abstract** keyword is a non-access modifier, used for classes and methods:

- Abstract class
  - is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).

- Abstract method:
  - can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

Abstract Classes and Methods

- An abstract class can have both abstract and regular methods:

```
1 abstract class Animal {
2   public abstract void animalSound();
3   public void sleep() {
4     System.out.println("Zzz");
5   }
6 }
```

Animal myObj = new Animal(); // will generate an error

Abstract Classes and Methods

- To access the abstract class, it must be inherited from another class.
- Let's convert the Animal class we used in the Polymorphism chapter to an abstract class:

```java
// Abstract class
abstract class Animal {
  // Abstract method (does not have a body)
  public abstract void animalSound();
  // Regular method
  public void sleep() {
    System.out.println("Zzz");
  }
}

// Subclass (inherit from Animal)
class Pig extends Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The pig says: wee wee");
  }
}

class Code0821 {
  public static void main(String[] args) {
    Pig myPig = new Pig(); // Create a Pig object
    myPig.animalSound();
    myPig.sleep();
  }
}
```

Java Interface

- Interfaces
  - An interface is a completely "abstract class" that is used to group related methods with empty bodies:

```
// interface
interface Animal {
  public void animalSound(); // interface method (does not have a body)
  public void run(); // interface method (does not have a body)
}
```

  - To access the interface methods,
    - the interface must be "**implemented**" (kinda like inherited) by another class with the implements keyword (instead of extends).
    - The body of the interface method is provided by the "implement" class:

Java Interface

```java
// Interface
interface Animal {
  public void animalSound(); // interface method (does not have a body)
  public void sleep(); // interface method (does not have a body)
}

// Pig "implements" the Animal interface
class Pig implements Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The pig says: wee wee");
  }
  public void sleep() {
    // The body of sleep() is provided here
    System.out.println("Zzz");
  }
}

class Main {
  public static void main(String[] args) {
    Pig myPig = new Pig();  // Create a Pig object
    myPig.animalSound();
    myPig.sleep();
  }
}
```

Java Interface

- Like **abstract classes**, **interfaces** *cannot be used to create objects*
- Interface methods do not have a body - the body is provided by the "implement" class
- On implementation of an interface, you must override all of its methods
- Interface methods are by default abstract and public
- Interface attributes are by default public, static and final
- An interface *cannot contain a constructor*

Java Interface

- Why And When To Use Interfaces?

1. To achieve security
   - hide certain details and only show the important details of an object (interface).

2. Java does not support "multiple inheritance"
   - a class can only inherit from one superclass.
   - However, it can be achieved with interfaces, because the class can implement multiple interfaces.
   - Note: To implement multiple interfaces, separate them with a comma

Multiple Interfaces

- To implement multiple interfaces, separate them with a comma:

```java
interface FirstInterface {
  public void myMethod(); // interface method
}

interface SecondInterface {
  public void myOtherMethod(); // interface method
}

class DemoClass implements FirstInterface, SecondInterface {
  public void myMethod() {
    System.out.println("Some text..");
  }
  public void myOtherMethod() {
    System.out.println("Some other text...");
  }
}

class Main {
  public static void main(String[] args) {
    DemoClass myObj = new DemoClass();
    myObj.myMethod();
    myObj.myOtherMethod();
  }
}
```

Java Anonymous Class

- An **anonymous class** is a class without a name. It is created and used at the same time.
- You often use anonymous classes to **override methods** of an existing class or interface, without writing a separate class file.
- Here, we create an anonymous class that extends another class and overrides its method:

Java Anonymous Class

```java
// Normal class
class Animal {
  public void makeSound() {
    System.out.println("Animal sound");
  }
}

public class Main {
  public static void main(String[] args) {
    // Anonymous class that overrides makeSound()
    Animal myAnimal = new Animal() {
      public void makeSound() {
        System.out.println("Woof woof");
      }
    }; // semicolon is required to end the line of code that creates the object

    myAnimal.makeSound();
  }
}
```

Java Anonymous Class

- Anonymous Class from an Interface
  - You can also use an anonymous class to implement an interface on the fly:

```java
// Interface
interface Greeting {
  void sayHello();
}

public class Main {
  public static void main(String[] args) {
    // Anonymous class that implements Greeting
    Greeting greet = new Greeting() {
      public void sayHello() {
        System.out.println("Hello, World!");
      }
    };

    greet.sayHello();
  }
}
}
```

Java Anonymous Class

- When to Use Anonymous Classes?
  - Use anonymous classes when you need to create a short class for one-time use. For example:
    - Overriding a method without creating a new subclass
    - Implementing an interface quickly
    - Passing small pieces of behavior as objects

Java Enums

- An enum is a special "class" that represents a group of constants
  - unchangeable variables, like final variables
- To create an enum, use the enum keyword (instead of class or interface), and separate the constants with a comma.
  - Note that they should be in uppercase letters:

```
enum Level {
  LOW,
  MEDIUM,
  HIGH
}
```

Java Enums

```java
public class Code0831 {
  enum Level {
    LOW,
    MEDIUM,
    HIGH
  }

  public static void main(String[] args) {
    Level myVar = Level.MEDIUM;
    System.out.println(myVar);
  }
}
```

Java Enums

- Enum in a Switch Statement

```java
enum Level {
  LOW,
  MEDIUM,
  HIGH
}

public class Main {
  public static void main(String[] args) {
    Level myVar = Level.MEDIUM;

    switch(myVar) {
      case LOW:
        System.out.println("Low level");
        break;
      case MEDIUM:
         System.out.println("Medium level");
        break;
      case HIGH:
        System.out.println("High level");
        break;
    }
  }
}
```

Java Enums

- Loop Through an Enum

```
for (Level myVar : Level.values()) {
  System.out.println(myVar);
}
```

```
LOW
MEDIUM
HIGH
```

User Input (Scanner)

- User Input(keyboard)
  - The Scanner class is used to get user input, and it is found in the java.util package.

  - To use the Scanner class,
    1. create an object of the class
    2. use any of the available methods found in the Scanner class documentation.
    3. In our example, we will use the nextLine() method, which is used to read Strings:

User Input (Scanner)

```java
import java.util.Scanner;  // Import the Scanner class

class Main {
  public static void main(String[] args) {
    Scanner myObj = new Scanner(System.in);  // Create a Scanner object
    System.out.println("Enter username");

    String userName = myObj.nextLine();  // Read user input
    System.out.println("Username is: " + userName);  // Output user input
  }
}
```

User Input (Scanner)

- Input Types
  - In the example above, we used the nextLine() method, which is used to read Strings. To read other types, look at the table below:

Method	Description
`nextBoolean()`	Reads a `boolean` value from the user
`nextByte()`	Reads a `byte` value from the user
`nextDouble()`	Reads a `double` value from the user
`nextFloat()`	Reads a `float` value from the user
`nextInt()`	Reads a `int` value from the user
`nextLine()`	Reads a `String` value from the user
`nextLong()`	Reads a `long` value from the user
`nextShort()`	Reads a `short` value from the user

User Input (Scanner)

Method
nextBoolean()
nextByte()
nextDouble()
nextFloat()
nextInt()
nextLine()
nextLong()
nextShort()

```java
import java.util.Scanner;

class Main {
  public static void main(String[] args) {
    Scanner myObj = new Scanner(System.in);

    System.out.println("Enter name, age and salary:");

    // String input
    String name = myObj.nextLine();

    // Numerical input
    int age = myObj.nextInt();
    double salary = myObj.nextDouble();

    // Output input by user
    System.out.println("Name: " + name);
    System.out.println("Age: " + age);
    System.out.println("Salary: " + salary);
  }
}put
  }
}
```