Classes and Objects

(Part 3, Nested Classes and Enumerated Types)

Reading:

• Java Tutorial: Section *Classes and Objects* under section *Learning the Java Language*
• Textbook: Chapter 6 (Page 368)
Nested Classes:

The Java programming language allows you to define a class within another class. Such a class is called a *nested class*.

```java
class OuterClass {
    ...
    class NestedClass {
        ...
    }
}
```
Terminology: Nested classes are divided into two categories: static and non-static. Nested classes that are declared static are called static nested classes. Non-static nested classes are called inner classes.

class OuterClass {
    ...
    static class StaticNestedClass {
        
    }
    class InnerClass {
        
    }
}

A nested class is a member of its enclosing class.

Non-static nested classes (inner classes) have access to other members of the enclosing class, even if they are declared private.

Static nested classes do not have access to other members of the enclosing class.
Why Use Nested Classes?

Compelling reasons for using nested classes include the following:

- **It is a way of logically grouping classes that are only used in one place:** If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.

- **It increases encapsulation:** Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.

- **It can lead to more readable and maintainable code:** Nesting small classes within top-level classes places the code closer to where it is used.
Static Nested Classes

As with class methods and variables, a static nested class is associated with its outer class.

Like static class methods, a static nested class cannot refer directly to instance variables or methods defined in its enclosing class.

---

**Note:** In effect, a static nested class is behaviorally a top-level class that has been nested in another top-level class for packaging convenience.

---

Static nested classes are accessed using the enclosing class name:

**OuterClass.StaticNestedClass**

For example, to create an object for the static nested class, use this syntax:

```java
OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();
```
**Inner Classes**

As with instance methods and variables, an inner class is associated with an instance of its enclosing class and has direct access to that object's methods and fields.

Also, because an inner class is associated with an instance, it cannot define any static members itself.

Objects that are instances of an inner class exist *within* an instance of the outer class. Consider the following classes:

```java
class OuterClass {
    
    class InnerClass {
    
    }
}
```

An instance of `InnerClass` can exist only within an instance of `OuterClass` and has direct access to the methods and fields of its enclosing instance.

**To instantiate an inner class, you must first instantiate the outer class.** Then, create the inner object within the outer object with this syntax:

```java
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

There are two special kinds of inner classes: local classes and anonymous classes.
Shadowing

If a declaration of a type (such as a member variable or a parameter name) in a particular scope (such as an inner class or a method definition) has the same name as another declaration in the enclosing scope, then the declaration shadows the declaration of the enclosing scope.

You cannot refer to a shadowed declaration by its name alone. The following example, `ShadowTest`, demonstrates this:

```java
public class ShadowTest {
    public int x = 0;

    class FirstLevel {
        public int x = 1;

        void methodInFirstLevel(int x) {
            System.out.println("x = " + x);
            System.out.println("this.x = " + this.x);
            System.out.println("ShadowTest.this.x = "+
                                ShadowTest.this.x);
        }
    }

    public static void main(String... args) {
        ShadowTest st = new ShadowTest();
        ShadowTest.FirstLevel fl = st.new FirstLevel();
        fl.methodInFirstLevel(23);
    }
}
```

**OUTPUT:**

```
x = 23
this.x = 1
ShadowTest.this.x = 0
```
Serialization

To serialize an object means to convert its state to a byte stream so that the byte stream can be reverted back into a copy of the object.

Serialization of inner classes, including local and anonymous classes, is strongly discouraged (Why?)

Example

DataStructure.java

Modifiers

You can use the same modifiers for inner classes that you use for other members of the outer class.

For example, you can use the access specifiers private, public, and protected to restrict access to inner classes, just as you use them to restrict access do to other class members.
Local and Anonymous Classes

There are two additional types of inner classes.

- You can declare an inner class within the body of a method. These classes are known as **local classes**.
- You can also declare an inner class within the body of a method without naming the class. These classes are known as **anonymous classes**.

Local Classes

Local classes are classes that are defined in a *block*, which is a group of zero or more statements between balanced braces.

You typically find local classes defined in the body of a method.

Declaring Local Classes

You can define a local class inside any block. For example, you can define a local class in a method body, a *for* loop, or an *if* clause.

Example: LocalClassExample.java in tutorial.
Accessing Members of an Enclosing Class

A local class has access to the members of its enclosing class.

However, starting in Java SE 8, a local class can access local variables and parameters of the enclosing block that are final or effectively final.

A variable or parameter whose value is never changed after it is initialized is effectively final.

For example, suppose that the variable `numberLength` is not declared final, and you add the highlighted assignment statement in the `PhoneNumber` constructor (See the `LocalClassExample` in tutorial):

```java
//From LocalClassExample
PhoneNumber(String phoneNumber) {
    numberLength = 7;
    String currentNumber = phoneNumber.replaceAll(
        regularExpression, "");
    if (currentNumber.length() == numberLength)
        formattedPhoneNumber = currentNumber;
    else
        formattedPhoneNumber = null;
}
```

Because of this assignment statement, the variable `numberLength` is not effectively final anymore. As a result, the Java compiler generates an error message similar to "local variables referenced from an inner class must be final or effectively final" where the inner class `PhoneNumber` tries to access the `numberLength` variable:

```java
if (currentNumber.length() == numberLength)
```
Local Classes Are Similar To Inner Classes

- Local classes are similar to inner classes because they cannot define or declare any static members.
- Local classes in static methods can only refer to static members of the enclosing class.
- Local classes are non-static because they have access to instance members of the enclosing block.
- You cannot declare static initializers or member interfaces in a local class.
- A local class can have static members provided that they are constant variables.

Anonymous Classes

Anonymous classes enable you to make your code more concise.

They enable you to declare and instantiate a class at the same time.

They are like local classes except that they do not have a name.

Use them if you need to use a local class only once.

An anonymous inner class is a subclass, or implements an interface.
Syntax of Anonymous Classes

An anonymous class is an expression. The syntax of an anonymous class expression is like the invocation of a constructor, except that there is a class definition contained in a block of code.

```java
public class HelloWorldAnonymousClasses {

    interface HelloWorld {
        public void greet();
        public void greetSomeone(String someone);
    }

    public void sayHello() {

        class EnglishGreeting implements HelloWorld {
            String name = "world";
            public void greet() {
                greetSomeone("world");
            }
            public void greetSomeone(String someone) {
                name = someone;
                System.out.println("Hello "+ name);
            }
        }

        HelloWorld englishGreeting = new EnglishGreeting();

        HelloWorld frenchGreeting = new HelloWorld() {
            String name = "tout le monde";
            public void greet() {
                greetSomeone("tout le monde");
            }
            public void greetSomeone(String someone) {
                name = someone;
            }
        }

    }
}
```
```java
System.out.println("Salut " + name);
}
};

HelloWorld spanishGreeting = new HelloWorld() {
    String name = "mundo";
    public void greet() {
        greetSomeone("mundo");
    }
    public void greetSomeone(String someone) {
        name = someone;
        System.out.println("Hola, " + name);
    }
};
englishGreeting.greet();
frenchGreeting.greetSomeone("Fred");
spanishGreeting.greet();
}

class HelloWorldAnonymousClasses {
    public static void main(String... args) {
        HelloWorldAnonymousClasses myApp =
            new HelloWorldAnonymousClasses();
        myApp.sayHello();
    }
}
```

The anonymous class expression consists of the following:

- The `new` operator
- The name of an interface to implement or a class to extend. In this example, the anonymous class is implementing the interface `HelloWorld`.
- Parentheses that contain the arguments to a constructor, just like a normal class instance creation expression. **Note:** When you implement an interface, there is no constructor,
so you use an empty pair of parentheses, as in this example.

- A body, which is a class declaration body. More specifically, in the body, method declarations are allowed but statements are not.

An anonymous class cannot have constructors, but can have instance initializer. Recall that an instance initializer is a block of initialization code contained within curly braces inside a class definition.

Because an anonymous class definition is an expression, it must be part of a statement. In this example, the anonymous class expression is part of the statement that instantiates the `frenchGreeting` object. (This explains why there is a semicolon after the closing brace.)

**Accessing Local Variables of the Enclosing Scope, and Declaring and Accessing Members of the Anonymous Class**

Like local classes, anonymous classes can capture variables; they have the same access to local variables of the enclosing scope:

- An anonymous class has access to the members of its enclosing class.
- An anonymous class cannot access local variables in its enclosing scope that are not declared as `final` or effectively final.
• Like a nested class, a declaration of a type (such as a variable) in an anonymous class shadows any other declarations in the enclosing scope that have the same name.

Anonymous classes also have the same restrictions as local classes with respect to their members:

• You cannot declare static initializers or member interfaces in an anonymous class.
• An anonymous class can have static members provided that they are constant variables.

Note that you can declare the following in anonymous classes:

• Fields
• Extra methods
• Instance initializers (but no constructors)
• Local classes

Enum Types

An enum type is a special data type that enables for a variable to be a set of predefined constants.

Common examples include compass directions (values of NORTH, SOUTH, EAST, and WEST) and the days of the week.

Because they are constants, the names of an enum type's fields are in uppercase letters.
In the Java programming language, you define an enum type by using the `enum` keyword.

```java
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}
```

You should use enum types any time you need to represent a fixed set of constants. That includes natural enum types such as the planets in our solar system and data sets where you know all possible values at compile time—for example, the choices on a menu, command line flags, and so on.

Here is some code that shows you how to use the `Day` enum defined above:

```java
public class EnumTest {
    Day day;

    public EnumTest(Day day) {
        this.day = day;
    }

    public void tellItLikeItIs() {
        switch (day) {
            case MONDAY:
                System.out.println("Mondays are bad.");
                break;

            case FRIDAY:
                System.out.println("Fridays are better.");
                break;

            case SATURDAY: case SUNDAY:
                System.out.println("Weekends are best.");
                break;
        }
    }
}
```
System.out.println("Midweek days are so-so.");
break;
}
}

public static void main(String[] args) {
    EnumTest firstDay = new EnumTest(Day.MONDAY);
    firstDay.tellItLikeItIs();
    EnumTest thirdDay = new EnumTest(Day.WEDNESDAY);
    thirdDay.tellItLikeItIs();
    EnumTest fifthDay = new EnumTest(Day.FRIDAY);
    fifthDay.tellItLikeItIs();
    EnumTest sixthDay = new EnumTest(Day.SATURDAY);
    sixthDay.tellItLikeItIs();
    EnumTest seventhDay = new EnumTest(Day.SUNDAY);
    seventhDay.tellItLikeItIs();
}

The output is:

Mondays are bad.
Midweek days are so-so.
Fridays are better.
Weekends are best.
Weekends are best.

Java programming language enum types are much more powerful than their counterparts in other languages. The enum declaration defines a class (called an enum type).

The enum class body can include methods and other fields.

The compiler automatically adds some special methods when it creates an enum. For example, they have a static values method that returns
an array containing all of the values of the enum in the order they are declared.

```java
for (Planet p : Planet.values()) {
    System.out.printf("Your weight on %s is %f\n", p, p.surfaceWeight(mass));
}
```

**Note:** All enums implicitly extend `java.lang.Enum`. Because a class can only extend one parent (see [Declaring Classes](#)), the Java language does not support multiple inheritance of state (see [Multiple Inheritance of State, Implementation, and Type](#)), and therefore an enum cannot extend anything else.
Note: The constructor for an enum type must be package-private or private access. It automatically creates the constants that are defined at the beginning of the enum body. You cannot invoke an enum constructor yourself.

```java
public enum Planet {
    MERCURY (3.303e+23, 2.4397e6),
    VENUS   (4.869e+24, 6.0518e6),
    EARTH   (5.976e+24, 6.37814e6),
    MARS    (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27,    7.1492e7),
    SATURN  (5.688e+26, 6.0268e7),
    URANUS  (8.686e+25, 2.5559e7),
    NEPTUNE (1.024e+26, 2.4746e7);

    private final double mass;   // in kilograms
    private final double radius; // in meters
    public Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }
    private double mass() { return mass; }
    private double radius() { return radius; }

    // universal gravitational constant  (m3 kg-1 s-2)
    public static final double G = 6.67300E-11;
    public double surfaceGravity() {
        return G * mass / (radius * radius);
    }
    public double surfaceWeight(double otherMass) {
        return otherMass * surfaceGravity();
    }

    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("Usage: java Planet <earth_weight> ");
            System.exit(-1);
        }
        double earthWeight = Double.parseDouble(args[0]);
    }
```
double mass = earthWeight/EARTH.surfaceGravity();
for (Planet p : Planet.values())
    System.out.printf("Your weight on %s is %f\n", p, p.surfaceWeight(mass));
}

If you run Planet.class from the command line with an argument of 175, you get this output:

$ java Planet 175
Your weight on MERCURY is 66.107583
Your weight on VENUS is 158.374842
Your weight on EARTH is 175.000000
Your weight on MARS is 66.279007
Your weight on JUPITER is 442.847567
Your weight on SATURN is 186.552719
Your weight on URANUS is 158.397260
Your weight on NEPTUNE is 199.207413