Chapter 7 - Methods and Encapsulation

Access Modifiers

Java offers four choices of access modifier:

- ***private*** The `private` modifier means the method can be called only from within the same class.
- **Default (Package-Private) Access** With default access, the method can be called only from classes in the same package.
- ***protected*** The `protected` modifier means the method can be called only from classes in the same package or subclasses. The `protected` rules apply under two scenarios

```
1:  package pond.swan;
2:  import pond.shore.Bird;     // in different package than Bird
3:  public class Swan extends Bird {     // but subclass of Bird
4:     public void swim() {
5:        floatInWater();              // subclass access to superclass
6:        System.out.println(text);    // subclass access to superclass
7:     }
8:     public void helpOtherSwanSwim() {
9:        Swan other = new Swan();
10:       other.floatInWater();        // subclass access to superclass
11:       System.out.println(other.text);  // subclass access
12:                                        // to superclass
13:    }
14:    public void helpOtherBirdSwim() {
15:       Bird other = new Bird();
16:       other.floatInWater();               // DOES NOT COMPILE
17:       System.out.println(other.text);     // DOES NOT COMPILE
18:    }
19: }
```

  - A member is used without referring to a variable. This is the case on lines 5 and 6. In this case, we are taking advantage of inheritance and `protected` access is allowed.
  - A member is used through a variable. This is the case on lines 10, 11, 16, and 17. In this case, the rules for the reference type of the variable are what matter. If it is a subclass, `protected` access is allowed. This works for references to the same class or a subclass.
- ***public*** The `public` modifier means the method can be called from any class.

Optional Specifiers

Unlike with access modifiers, you can have multiple specifiers in the same method (although not all combinations are legal). When this happens, you can specify them in any order.

- ***static*** The `static` modifier is used for class methods.
- ***abstract*** The `abstract` modifier is used when a method body is not provided.
- ***final*** The `final` modifier is used when a method is not allowed to be overridden by a subclass.
- ***synchronized*** The `synchronized` modifier is used with multithreaded code.
- ***native*** The `native` modifier is used when interacting with code written in another language such as C++.
- ***strictfp*** The `strictfp` modifier is used for making floating-point calculations portable.

Method Name

Method names follow the same rules as we practiced with variable names.

- An identifier may only contain letters, numbers, `$`, or `_`.
- The first character is not allowed to be a number, and reserved words are not allowed.
- Single underscore character is not allowed.
- By convention, methods begin with a lowercase letter but are not required to.

Working with Varargs

- A varargs parameter must be the last element in a method's parameter list.
- You are allowed to have only one varargs parameter per method.

Designing *static* Methods and Fields

Assuming that count is a static variable

```
5: Koala k = new Koala();
6: System.out.println(k.count);          // k is a Koala
7: k = null;
8: System.out.println(k.count);          // k is still a Koala
```

Believe it or not, this code outputs 0 twice. Line 6 sees that `k` is a `Koala` and `count` is a `static` variable, so it reads that `static` variable. Line 8 does the same thing. Java doesn't care that `k` happens to be `null`. Since we are looking for a `static`, it doesn't matter.

A `static` method or instance method can call a `static` method because `static` methods don't require an object to use. Only an instance method can call another instance method on the same class without using a reference variable, because instance methods do require an object. Similar logic applies for the instance and `static` variables.

```
public class Giraffe {
   public void eat(Giraffe g) {}
   public void drink() {};
   public static void allGiraffeGoHome(Giraffe g) {}
   public static void allGiraffeComeOut() {}
}
```

Type	Calling	Legal?
`allGiraffeGoHome()`	`allGiraffeComeOut()`	Yes
`allGiraffeGoHome()`	`drink()`	No
`allGiraffeGoHome()`	`g.eat()`	Yes
`eat()`	`allGiraffeComeOut()`	Yes
`eat()`	`drink()`	Yes
`eat()`	`g.eat()`	Yes

Passing Data among Methods

- Java uses pass-by-value to get data into a method.
- Assigning a new primitive or reference to a parameter doesn't change the caller.
- Calling methods on a reference to an object can affect the caller.

Overloading Methods

- *Method overloading* occurs when methods have the same name but different method signatures, which means they differ by method parameters.

```
public void fly(int numMiles) {}
public void fly(short numFeet) {}
public boolean fly() { return false; }
void fly(int numMiles, short numFeet) {}
public void fly(short numFeet, int numMiles) throws Exception {}
```

-

```
public void fly(int numMiles) {}
public int fly(int numMiles) {}     // DOES NOT COMPILE

public void fly(int[] lengths) {}
public void fly(int... lengths) {}     // DOES NOT COMPILE
```

Autoboxing

-

```
public void fly(int numMiles) {}
public void fly(Integer numMiles) {}
```

calling `fly(3)` will match the `int numMiles` version. Java tries to use the most specific parameter list it can find. When the primitive `int` version isn't present, it will autobox. However, when the primitive `int` version is provided, there is no reason for Java to do the extra work of autoboxing.

Generics

You might be surprised to learn that these are not valid overloads:

```
public void walk(List<String> strings) {}
public void walk(List<Integer> integers) {}    // DOES NOT COMPILE
```

Java has a concept called *type erasure* where generics are used only at compile time

Putting It All Together