Head First Java: Chapter 4 Notes

August 27, 2019

Object methods

Objects created from the same class will have the same instance variables (but not the same values) and the same methods as described by the class. However, the methods might behave differently across objects from the same class due to method dependence on its instant variables' values. For example, a method for a *Dog* class might have a method `bark()` that has a conditional statement that depends on the instance variable `size`.

Like functions (and methods) in other programming language, Java object's methods can use parameters by passing arguments when calling them. A method for the class *Dog* can be written to accept arguments or not, here is one that does not accept arguments:

```
void bark() {
    System.out.println(‘‘ruff’’);
}
```

In this case to have the *Dog* bark multiple times you would call the method multiple times outside the code of the method itself. However, you can also write the `bark()` method to accept an integer argument:

```
void bark(int numOfBarks) {
    while (numOfBarks > 0) {
        System.out.println(‘‘ruff’’);
        numOfBarks = numOfBarks - 1;
    }
}
```

In this version of `bark()` the user will specify how many times they want the *Dog* to bark by passing an argument to it. Methods can accept multiple arguments if it was defined to have the same number of parameters:

```
void takeTwo(int x, int y) {
    int z = x + y;
    System.out.println(‘‘Total is ’’ + z);
}
```

Note that when passing arguments to a method, they can be values or variables, as long as the type matches. So:

```
takeTwo(34, -55);
int spam = 45;

int egg = 99;
takeTwo(spam, egg)
```

Both method calls are valid because the arguments are of the right type. When passing variables as arguments to methods, take note that any changes to the variable passed as argument does **NOT** change the value of the variable itself. That is:

```
int x = 7;
int go(int z){
    z = 0;
    return z
}
System.out.println(foo.go(x));
System.out.println(x);
```

Here `go()` is a method that takes any integer and returns 0. An integer variable `x` with the value of `7` was passed as an argument to `go()`, and thus when the `println()` method was called `0` was printed to the screen. However, then `println()` was called again to print `x` to the screen it will print `7`. The `go()` method only made a **COPY** of the value of `x` and never modified the value of `x` itself. In Java, all arguments are *passed-by-copy* (or: passed by value).

In Java, distinguishing a method that returns a value is not trivial and must be declared at the beginning. Java methods **MUST** declare return type, even when no return value is desired; in that case `void` needs to be declared. A `void` method returns no value. The return value of a method is type limited, depending on what was declared in the beginning. For example:

```
int giveSecret() {
    return 42;
}
```

This method is valid and compatible because it was declared to return an `int` value, and it did (`42`). To return multiple values of the same type, a method can be declared with an array argument.

Object Encapsulation

Instance variables that can be freely accessed and modified are dangerous and lead to unexpected behavior or malicious attack. For example, malicious users can modify an exposed instance variable by `car.velocity = -20;`, which would presumably crash a program. Thus, it is good practice to *encapsulate* all object instance variables by flagging them `private`, and add `public` mutator methods for the appropriate access. Consider:

```
class GoodDog {
    private int size;
    public int getSize() {
        return size;
    }
    public void setSize(int s) {
    size = s;
    }
}
```

Methods and other codes outside of `GoodDog` cannot access or modify `GoodDog.size` because instance variable is set to private. A public mutator method — `GoodDog.setSize()` — to set the value of `GoodDog.size`.