Java OO Concepts Unit

Lesson 01.2: Classes, Objects, and Creating New Types

SOFTWARE GUILD

Lesson 01.2: Classes, Objects, and Creating New Types

Overview

In this portion of Lesson 01, we'll look at how we define and create new type in Java and what makes up these new types.

Creating New Types

Every time we define a new class in Java, we are defining a new type. As discussed earlier in the course, there are two categories of data types in Java: **native types** and **user-defined types** — new classes fall into the latter category.

Types (classes) in Java simply consist of **fields** (or **properties**) and **behaviors** (or **methods**). We've used several user-defined types in our programs already; for example, Scanner and String.

Classes vs. Objects

A class is a definition, like the blueprint of a house. A blueprint is a detailed model of a building — it may show you how to build your house, but you can't live in a blueprint. You have to build the house, following the plan in the blueprint, before you can move in. Similarly, you must **instantiate** an object, based on the definition contained in the class, before you can use it.

Another way to approach this is to think of a class as an idea and an object as the instantiation of that idea. For example, a class is like the idea of a German shepherd, whereas an object is my German shepherd named Buster. You can pet Buster, but you can't pet the idea of a German shepherd.

Properties, Accessors, and Mutators

A common technique used to achieve data hiding in Java is the use of **accessors** and **mutators** (these are also known as **getters** and **setters** in Java). Accessors and mutators are simply methods that get and set (respectively) the values of the properties (or fields) on an object.

So, why would we go to all of the trouble to create these methods when we could simply let clients access and change the values of our class's properties directly? Accessors and mutators assist us with data hiding — the client has no idea how the properties are stored or calculated, he/she just knows what each accessor/mutator does. For example, a Student class might have a property called gradePointAverage. If we use accessors and mutators, we are free to store gradePointAverage as a single value or we might choose to calculate gradePointAverage every time the accessor is called. If we chose to do that, we would most likely want to make gradePointAverage a **read-only** field. Accessors and mutators help us here as well; to make a property read-only, we simply don't implement a mutator method for that property.

Methods/Behaviors

In addition to properties (and their corresponding accessors and mutators), classes can have behaviors. The behaviors of a class are implemented as **methods**. As we saw earlier in the course, methods are simply named blocks of code that can be **invoked** (or **called**) by other blocks of code in order to accomplish some purpose.

Constructors

A **constructor** is a special method that is called when you create an instance of your class. Constructors are usually used to initialize the properties of newly-instantiated object. Although constructors are methods, there are some special rules that must be followed when creating a constructor:

1. A constructor must have the same name as the class that it is a part of. For example, the constructor for a class called **Dog** would be **Dog()**.
2. Constructors never have a return type, not even void.
3. Constructors can have parameters but don't have to.
4. There can be more than one constructor in a given class.
5. You don't have to create a constructor for your class. If you don't create one, the JVM will supply one called the **default constructor**. This constructor has no parameters and is simply an empty block of code.

An Example

The following is an example of a class that models a dog.

Notes:

1. Name is a read-only property. After it is set in the constructor, there is no way to change the value.
2. Age and weight are read/write properties

```
public class Dog {
    private String dogName;
    private int age;
    private int weight;

    public Dog(String nameIn, int ageIn, int weightIn) {
        dogName = nameIn;
        age = ageIn;
        weight = weightIn;
    }
    public String getName() {
        return dogName;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int ageIn) {
        this.age = ageIn;
    }
    public int getWeight() {
        return weight;
    }
    public void setWeight(int weightIn) {
        this.weight = weightIn;
    }
```

```
    public void bark() {
        System.out.println("WOOF");
    }
    public void sit() {
        System.out.println("Sitting...");
    }
}
```

Static

Now that we know a little bit more about classes and objects, we'll revisit the **static** keyword:

1. If a property of method is marked as static, it means that it is associated with the **class** and not with any particular instantiation of the class.
2. Static properties and methods can be accessed without creating an instance of the class.
3. Non-static properties and methods are associated with a particular instantiation of the class, which means that they are only accessible through an instantiated object.

Dot Operator

The **dot operator** (.) is used to access public properties or methods of an object. The dot operator is used for static and non-static properties and methods. We have seen many examples of the dot operator:

- Math.abs(...) - static
- System.out.println(...) - static
- myDog.bark() - non-static

This Keyword

The **this** keyword is used to refer to the instance of the class which the code is currently executing. It is used in conjunction with the dot operator to access properties and methods of the containing class. It is common to see the this keyword used in accessors and mutators (see the Dog example above).