Lesson 02: Classes, Objects, and Creating New Types

Overview

In this lesson, we'll look at how we define and create new types in Java and what comprises these new types.

Objectives

By the end of this lesson, you will be able to:

1. Create new types.
2. Differentiate between classes and objects.
3. Explain the use of accessors and mutators (or getters and setters).
4. Use the dot operator to access object public properties or methods.
5. Use the `this` keyword.
6. Instantiate an object.
7. Invoke a method.
8. Relate constructors to methods.
9. Apply the static keyword to methods and constants.

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.

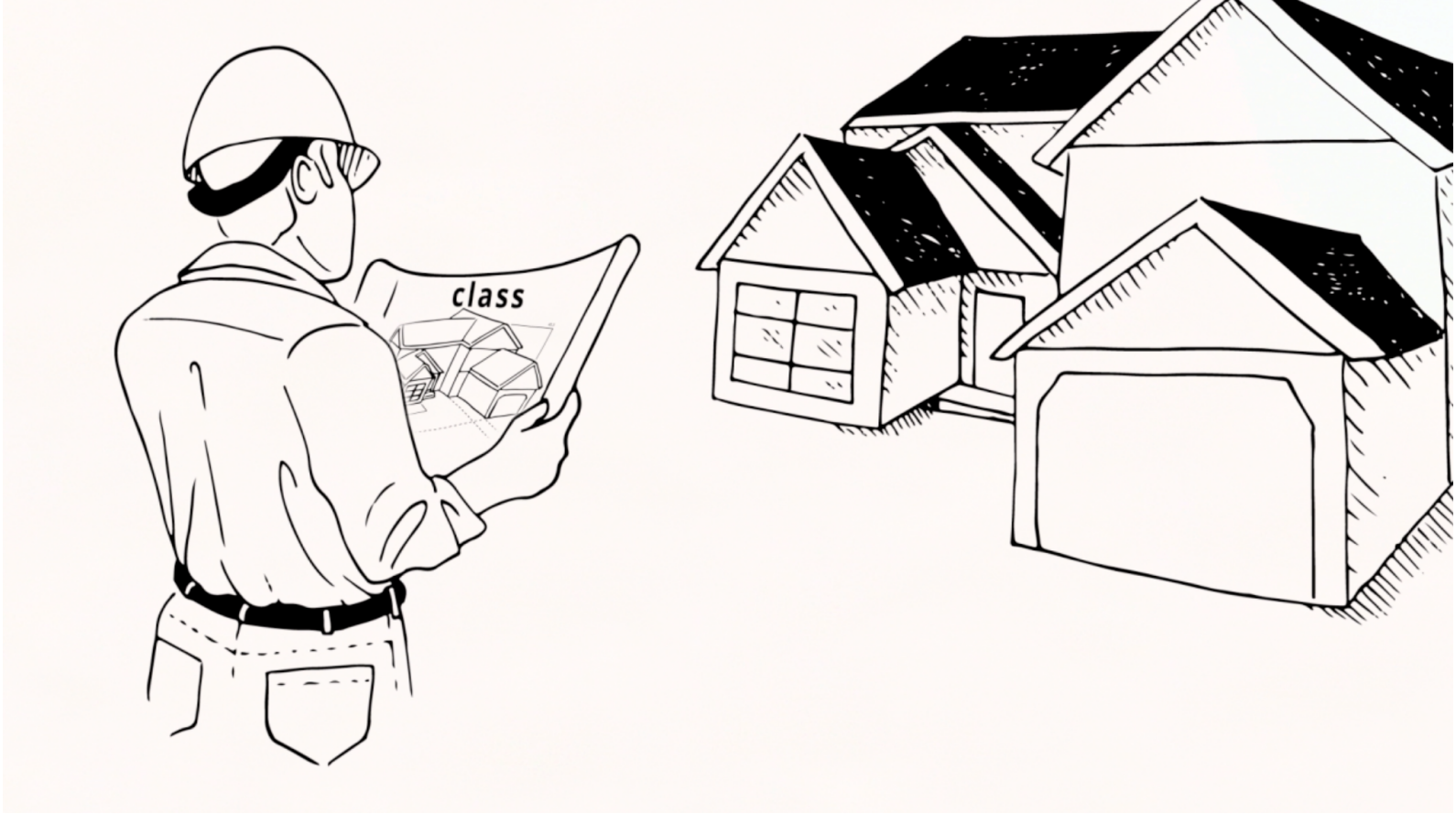
Types (classes) in Java simply consist of **fields** (or **properties**) and **behaviors** (or **methods**). Fields and behaviors are sometimes referred to as **members**. You have probably already used several user-defined types, including `Scanner` and `String`.

```
public class Dog {

    private String name;
    private double weight;

    public String getName() {
```

```
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getWeight() {
        return weight;
    }

    public void setWeight(double weight) {
        this.weight = weight;
    }

    public void bark() {
        System.out.println("WOOF!");
    }

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

}
```

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

If you recall our previous lesson, we talked about encapsulation and data hiding. 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 the trouble to create these methods when we could simply let clients access and change the values of our class's properties directly? As mentioned before, it is desirable for code that uses an object to have no idea how the properties are stored or calculated. This code should just know what each getter and setter does. For example, a `Student` class might have a property called `gradePointAverage`. If we use getters and setters, we are free to store gradePointAverage as a single value or we might choose to calculate `gradePointAverage` every time the getter is called. If we chose to do that, we would most likely want to make `gradePointAverage` a read-only field. Getters and setters help us here as well; to make a property read-only, we simply don't create a setter method for that property.

In the example below, the `Student` class has two regular properties: `name` and `grades` and one calculated read-only property called `gradePointAverage`. Although we can see (by looking at the

code of the class) that `gradePointAverage` does not have a backing field like name and grades do, any code using the getter for gradePointAverage has no way to know that — it just sees a getter for `gradePointAverage`.

```
public class Student {
    private String name;
    private double[] grades = new double[4];

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double[] getGrades() {
        return grades;
    }

    public void setGrades(double[] grades) {
        this.grades = grades;
    }

    public double getGradePointAverage() {
        double totalPoints = 0;

        for(int i = 0; i < grades.length; i++) {
            totalPoints += grades[i];
        }
        return totalPoints/grades.length;
    }
}
```

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. On the left side of the dot operator is the class name (for static fields and methods) or the variable name of the instance (for non-static fields or methods). On the right side of the dot operator is the method or field we want to access.

We have seen many examples of the dot operator:

- `SimpleMath.add(...)` – static
- `System.out.println(...)` – static
- `myDog.bark()` – non-static
- `currentStudent.getGradePointAverage()` – non-static

`this` Keyword

The `this` keyword is used to refer to the instance of the class in 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, mutators, and constructors (see the Dog example above). Note that the `this` keyword can never be used inside a static method because static methods are never associated with any particular instance of a class.

Methods/Behaviors

In addition to properties (and their corresponding getters and setters), 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 code in the program in order to accomplish some purpose. Methods are always contained inside a class definition — they cannot stand on their own.

In the example below, you see both regular methods — `bark()` and `sit()` — and getter/setter methods for the name and weight properties:

```
public class Dog {

    private String name;
    private double weight;

    public String getName() {
```

```
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getWeight() {
        return weight;
    }

    public void setWeight(double weight) {
        this.weight = weight;
    }

    public void bark() {
        System.out.println("WOOF!");
    }

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

}
```

If your class is well-designed, the methods in the class will match the purpose of the class. In other words, they must be cohesive. For example, in the `Dog` class above, it would not make any sense to have a method called `meow()`.

Note that the methods on the `Dog` class are not marked static.

Up until this point, we have marked all methods static so this is something new. Non-static methods are known as **instance methods**. We look at object instantiation and how to invoke instance methods on an object later in this document. We take a detailed look at the static keyword and its use at the end of this lesson.

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 a newly-instantiated object. Although constructors are methods, there are some special rules that must be followed when creating a constructor:

- 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()`.
- Constructors never have a return type, not even void.
- Constructors can have parameters but don't have to.
- There can be more than one constructor in a given class.
- You don't have to create a constructor for your class. If you don't create one, the compiler will supply one called the **default constructor**. The default constructor has no parameters and the one that the JVM supplies simply has an empty method body. Such a constructor is the default constructor whether you write it or the compiler provides it.

Let's take another look at the `Dog` class — this time with a default constructor and one that takes `name` and `weight` parameters:

```
public class Dog {

    private String name;
    private double weight;

    public Dog() {

    }

    public Dog(String nameIn, double weightIn) {
        this.name = nameIn;
        this.weight = weightIn;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
```

```
        this.name = name;
    }

    public double getWeight() {
        return weight;
    }

    public void setWeight(double weight) {
        this.weight = weight;
    }

    public void bark() {
        System.out.println("WOOF!");
    }

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

}
```

Object Instantiation and Method Invocation

Because the `Dog` class has two constructors, we now have two options for creating a new Dog object. We can create a `Dog` using the default constructor that we wrote (i.e., the one that has no arguments) or we can use the constructor that has two parameters.

First, let's look at creating a `Dog` object using its default constructor. We'll start by creating a class called `App`. We'll include a `main` method from which we'll instantiate our `Dog` object.

```
public class App {

    public static void main(String[] args) {

        Dog myDog = new Dog();
```

```
    }
}
```

If no values are explicitly set on the fields of a class when it is instantiated, the fields get initialized to their default values: null for user-defined types, 0 for numbers, and false for booleans. Since we used the constructor with no arguments, the fields in our new `Dog` (`myDog`) were initialized to their default values: null for `name`, and 0.0 for `weight`.

In order to set values for `name` and `weight`, we must invoke the setter methods for these two properties. As mentioned above, we do this with the dot operator:

```
public class App {

    public static void main(String[] args) {

        Dog myDog = new Dog();
        myDog.setName("Spot");
        myDog.setWeight(34.0);

    }
}
```

Now `myDog` has the name Spot and weights 34 pounds.

Because we have the second constructor, we can create a new `Dog` with values of our choosing for name and weight when we instantiate the object, as in the following example:

```
public class App {

    public static void main(String[] args) {

        Dog anotherDog = new Dog("Buster", 23.5);

    }
}
```

The variable `anotherDog` is instantiated with the name `Buster` and a `weight` of 23.5 pounds.

Static

Now that we know a little bit more about classes and objects, let's revisit the `static` keyword to learn where and when it is appropriate to use this keyword for fields and methods. First, let's look at some facts about the `static` keyword:

- If a field or method is marked as `static`, it means that it is associated with the **class** and not with any particular instantiation of the class. This means that there is only ever one copy of a field or method that is marked static. This one copy exists whether zero or 97 instances of the class are created.
- Static fields and methods can be accessed without creating an instance of the class. This follows from the previous item. The one and only copy of a `static` field or method exists even if no instances of the class have been instantiated.
- 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. This also means that non-static fields and methods do not exist until one or more instances of the class have been created.

Given this curious set of properties, when is it appropriate or desirable to mark a method or property static? There are three common use cases for the `static` keyword:

1. Main Method

We have seen this example of the `static` keyword from the very first program we wrote: HelloWorld. The main method is the **entry point** of our program — this is where the program kicks off. Since this is the first thing that is going to be run in the program (even before any objects are instantiated!), it has to be static. There is no way for the JVM to create our object and then call main — where would that code live? Instead, the JVM locates our class and then calls main to begin execution.

2. Constant Values

Constants are another place where it is appropriate to use the static keyword. We can define a constant value (for example, pi) as a `static` field on a class. This means that there will only ever be one copy of that constant in our program. The value can't be changed, so there is no reason to have a copy of the value associated with every instance of the class. The example below shows what this looks like.

3. Utility Methods

Utility methods such as those performing math operations are great candidates for using the `static` keyword. Methods that are marked `static` must not attempt to change the state of the class with which they are associated. Again, math operations are perfect examples of this — they

take inputs, operate on those inputs and produce outputs. In fact, all of the methods and constants on the `Math` class in Java are marked static.

Let's take a look at a very simple class to see what a static constant and static methods look like:

```
public class SimpleMath {

    public static final double PI = 3.14;

    public static int add(int a, int b) {
        return a + b;
    }

    public static int subtract(int a, int b) {
        return a - b;
    }

}
```

As you can see, we have declared `PI` as a `public static final double` field and we have two methods, `add` and `subtract`, that are also marked `static`.

We have discussed why marking these fields and methods as `static` makes sense but we haven't discussed how this will affect how other code interacts with this field and these methods. Marking these members `static` means that we can access them without first instantiating an instance of the `SimpleMath` class. In some occurrences, we have to first instantiate an object before we can access its methods. One example is `Scanner`, which requires that we declare and instantiate a `Scanner` before we can call a method on our new instance, like this:

```
Scanner myScanner = new Scanner(System.in);
myScanner.nextLine();
```

When using `static` methods or fields, we do not have to first instantiate the object; we can use it directly from the class. This is because static members are associated with the class itself, rather than with a particular instantiation of the class:

```
SimpleMath.add(5, 3);
```