The Tiny Book of Objects

Sridhar Narayan

August 14, 2017

Chapter 1

Objects

1.1 Why Objects

Consider the task of writing a computer program that can prompt a user to provide two numbers. The program then adds the numbers provided and shows the result to the user. A little reflection yields a program that looks something like this (in pseudocode):

```
Prompt user for first number
Store user input in memory
Prompt user for the second number
Store user input in memory
Add user supplied numbers and store result in memory
Display result
```

Note that this program can be designed by focusing on the actions the *computer* must take in order to accomplish the goal, i.e. an *algorithmic* or *machine-centric* view. Now, consider the task of writing a computer program that functions as a media player, for example like *iTunes*. It is immediately evident that while the machine-centric view works well for small(er) problems, it is harder to think in terms of what instructions must be executed by the computer to accomplish a given goal, when the problem is more complex. Thus, for large(r) problems, the algorithmic approach is less appropriate. Object oriented programming is better suited to the demands of large(r) software projects.

1.2 What is an object?

An object is a named chunk of data ***and*** code. The data characterizes the objects' structure, while the code characterizes its behavior. The name provides a mechanism for referencing the object, i.e. its data and code.

1.2.1 Object Structure

All objects have properties (attributes) that describe the structure of the object. For instance, if you were creating a ball to be used in an animation, you may want to define the following properties for the ball: *radius, color, position (on the screen)*. Or, a fraction object may have the properties, *numerator* and *denominator*.

1.2.2 Object Behavior

Object behavior answers the question, "What operations are allowed on this object ?", or "To what messages does this object respond ?" For instance, the ball in the previous section may respond to the messages *move, shrink, grow, explode, changeColor etc.* A fraction object may allow operations like *add, multiply* etc.

1.3 What is a class

There may be many objects of a given kind. For instance, an animation may involve dozens of balls. While each ball can be individually described, it is easier to develop a generic description for a ball that can be repeatedly used to model each ball. A *class* is a *generic* description of an object, and all objects are *instances* of some class.

1. All objects of a given class have the same attributes. However, the values of these attributes need not be the same. Thus, all balls have a radius, but one ball may have a larger radius than another.
2. All objects of a given class have the same behavior, i.e they all respond to the same messages.

1.4 How is a class defined in Java

1.4.1 Defining the structure

The following Java code describes the structure of a ball.

```
public class Ball {

    private int radius;
    private Point position.
    protected Color color;
    public static String manufacturer = “Shanghai Ball Factory”;

}
```

This code fragment says that every instance of this class, i.e. every Ball has a radius, a color, and a position. These are known as *instance variables,* because the variables are specific to each instance of this class. **If one were to create 100 instances of Ball, each of those 100 Ball objects could have a different radius, color, and position.** The **static** keyword declares manufacturer to be a class variable. More on this below.

Java requires that each property be *declared*. Thus, each property must have a name. Also, the *type* of data associated with each property must be specified. Thus, radius is of type *int*(eger). Note that objects may have properties that are defined in terms of other objects. Thus, *position* is of type *Point*, where Point is itself a class. An examination of the Point class may reveal,

```
public class Point {

    private int x;
    private int y;

}
```

This shows that big objects can be defined in terms of smaller objects, which, in turn, may be defined in terms of tiny objects. This is a mechanism for managing complexity.

1.4.2 Defining Behavior

Here are two examples of *methods* or code that might be found in the class Ball.

```
public int getRadius() {

    return radius;

}
public int setRadius(int currentRadius) {

    radius = currentRadius;

}
public static String getManufacturer() {

    return manufacturer;

}
```

Note that methods often access object properties. Two questions arise:

1. Which object properties can a method access, i.e. read or modify? Answer: All properties defined in a class can be accessed by methods contained in *that* class. However, properties marked *private* can only be accessed by methods contained in that class. Such properties are not visible to methods defined in other classes. This *encapsulates* the data within the class. When a *private* datum changes, the source of the change is in a method in the class, a fact that helps troubleshoot malfunctioning programs. Properties marked *public* are visible in methods in other classes. More on this below.

2. Since each Ball has its own radius, which object’s radius does the *setRadius* method modify? Answer: Methods are invoked when an object receives the corresponding *message*. Thus, if a Ball named *bouncyBall* receives the message *setRadius,* the corresponding invocation of *setRadius* modifies the *bouncyBall’s* radius.

Also note that the two methods are labeled *public*. This says that these methods can be invoked from any method in any class, and this is a typical practice in object oriented programming. Properties are shielded from methods in other classes by labeling them *private. Public* methods then provide regulated, i.e. controlled, access to these properties.

1.5 Creating Objects

Objects are *instantiated* (created) by the *new* operator. The *main* method below, defined in a class named TestBall, demonstrates this.

```
public class TestBall {
    public static void main(String [] args) {
       Ball bouncyBall = new Ball();
       bouncyBall.color = Color.RED; //allowed because color is marked public in the class Ball
       bouncyBall.setRadius(20); //bouncyBall.radius = 20; is not legal because radius is privat
    }
}
```

The *main* method is the method that is **executed first** when a class is loaded into memory and executed by the Java Virtual Machine (JVM). However, this is true for only the first class encountered. The JVM may load and execute other classes in this process. For instance, the Ball class will be loaded before the *bouncyBall* is instantiated. Its *main* method, if defined, will not be executed.

1.6 Dot Notation and the keyword *static*

Note the use of the *dot* notation to reference methods and properties of objects. Thus, *instance variables* and *instance methods* can only be referenced using the form *objectName.propertyName* and *objectName.methodName*. Note also that the dot notation is used in the context of the **class Color**. The use of the form ***className****.propertyName* or ***className****.methodName* is only allowed for *class variables* and *class methods.* Class variables and methods are marked *static.* Class variables and methods differ from instance variables and methods in several important ways:

1. Class variables and methods exist even if no objects of that class exist. They belong to the class, which, of course, has an existence separate from its instances. Indeed, classes exist even if no instances of that class are in existence! Thus, Ball.getManufacturer() is legal at all times - even if no Ball has been instantiated.
2. There is only one copy of a class variable, whereas there are as many copies of instance variables as there are instances of that class.
3. Class variables and methods can also be accessed through object names, as in bouncyBall.getManufacturer(). The manner of reference, via the object or the corresponding class, is irrelevant since only ONE COPY of the class data exists.
4. Since class methods can be invoked without reference to any object of the class, they can only contain references to class variables. Violations of this rule will result in the familiar Java compiler error message, "Static reference to non-static variables..."

1.7 Inheritance

If you wish to define a new class BeachBall, you can define it much like the Ball class since BeachBall's share properties and behavior found in the class Ball. Inheritance provides a mechanism to avoid this obvious duplication.

```
public class BeachBall extends Ball {
}
```

Inheritance implies that a *BeachBall* ***is-A*** *Ball.* That is, a BeachBall is a kind of Ball. The Ball class is the *parent class*, while the BeachBall class is the child. The `extends` keyword confers upon BeachBall all the attributes and behavior of the class Ball. Thus, BeachBall's have radius, color, and a manufacturer's identity. Any methods defined for a Ball can be used with a BeachBall. New methods, like *inflate* below, and properties, like *numberOfPanels*, can be defined for BeachBall. However, these have no impact on the Ball class. A Ball cannot be *inflate*(d), and a Ball does not have the property *numberOfPanels*.

```
private int numberOfPanels;
public void inflate(int radiusChange) {

    radius = radius + radiusChange; //Illegal. Cannot access private variable radius.
}
public void changeColor(Color currentColor) {
    color = currentColor; //Legal. color is declared protected in Ball
}
```

The problem in the previous method will be immediately obvious to the perceptive reader. *radius,* being private in Ball, is not visible in the BeachBall class. On the other hand, *color,* which is declared *protected,* is visible in the child class. Visibility modifiers can thus be summarized.

- ***private*** - means only visible in the class in which it is defined - for internal use only.
- ***protected*** - means only visible in the class in which it is defined and in all its ***descendants*** - for family use only.
- ***public*** - means visible in all classes - anywhere and everywhere - for use by everyone.

1.8 Inheritance has implications

Since a BeachBall ***is a*** Ball, the following statement is legal.

```
Ball bigBouncyBall = new BeachBall();
```

This can be interpreted as follows. BeachBall is a *sub-type* of the *super-type* Ball. Without inheritance, variables have only one type, the one they are ***declared*** to have, i.e. when they are compiled. This is their *compile-time* or *static* type[1]. With inheritance, variables can have a second type, their *run-time or dynamic* type. For example, the static type of bigBouncyBall is Ball, whereas its dynamic type is BeachBall. In other words, the static type of a variable is what you said it would refer to, at compile time. The dynamic type of a variable is what it actually points to at run time. Thet static type of a variable remains fixed. However, its dynamic type can change as the program executes and is determined by the object to which the variable refers at any particular moment.

The following statement, however, is not legal.

```
BeachBall squishyBall = new Ball();
```

[1] Not to be confused with the Java keyword *static*. Same word, different meanings.

Index

algorithmic, 2

class, 2
class methods, 4
class variables, 4
compile-time, 5

declared, 3
dot notation, 4
dynamic, 5

Inheritance, 4
instance methods, 4
instance variables, 3
instances, 2
instantiated, 4
is-A, 4

machine-centric, 2
main method, 4
message, 3
methods, 3

private, 3
protected, 4
public, 3

run-time, 5

static, 3-5
sub-type, 5
super-type, 5