CHAPTER 5

Arrays, Lists, Sets, and Maps

So far, I've only talked about single values, but in programming, you often have to work with large collections of values. For this, we have many data structures that are built into the language. These are similar for Java, Groovy, Scala, and even JavaScript.

5.1 Arrays

An *array* is a fixed size collection of data values. Honestly, you probably won't use an array very often, but it's an important concept to learn.

You declare an array-type in Java by appending [] to the type. For example, an array of `int`s is defined as `int[]`.

```
int[] vampireAges = new  int[10]; // ten vampires
```

Accessing the values in an array uses the same square-bracket syntax, such as

```
vampireAges[0] = 1565; // first vampire
```

As you can see, the first index of an array is zero. Things tend to start at zero when programming; try to remember this.

Patient 0 Here's a helpful metaphor: the first person to start an outbreak (a zombie outbreak, for example) is known as patient zero, not patient one. Patient one is the *second* person infected.

This also means that the *last* index of the array is always one less than the size of the array. This is also true for lists.

```
vampireAges[9] = 442; // last vampire
```

You can reassign and access array values just like any other variable.

```java
int year = 2016; // current year?
int firstVampBornYear = year - vampireAges[0];
```

You can also declare arrays of objects as well. In this case, each element of the array is a reference to an object in memory.

```java
Vampire[] vampires = new Vampire[10]; // Vampire array with length 10
```

You can also populate your array directly, such as if you're creating an array of strings, for example.

```java
String[] names = {"Dracula", "Edward"};
```

Unfortunately, arrays are difficult to use in Groovy and the `Array` object in JavaScript is more like a Java `List`. Java arrays are a somewhat low-level structure, used only for performance reasons.

5.2 Lists

Of course, we don't always know how many elements we need to store in an array. For this reason (and many others), programmers invented `List`, a resizable collection of ordered elements.

In Java, you create `List` in the following way:

```java
List<Vampire> vampires = new  ArrayList<>();
```

The angle-brackets (`<>`) define the *generic type* of the list—what can go into the list. You can now add vampires to this list all day, and it will expand, as necessary, in the background.

You add to `List` like this:

```java
vampires.add(new  Vampire("Count Dracula", 1897));
```

`List` also contains tons of other useful methods, including:

- `size()`: Gets the size of `List`
- `get(int index)`: Gets the value at that index
- `remove(int index)`: Removes the value at that index
- `remove(Object o)`: Removes the given object
- `isEmpty()`: Returns `true` only if `List` is empty
- `clear()`: Removes all values from `List`

In Java, `List` has many different implementations, but we'll just focus on two (don't worry about the details here).

- `java.util.ArrayList`
- `java.util.LinkedList`

The only difference you should care about is that, in general, `LinkedList` grows faster, while `ArrayList`'s `get()` method is faster.

You'll learn how to loop through lists, arrays, and sets (and what "loop" means) in the next chapter. For now, just know that lists are a fundamental concept in programming.

5.2.1 Groovy Lists

Groovy has a simpler syntax for creating lists, which is built into the language.

```
def list = []
list.add(new Vampire("Count Dracula", 1897))
// or
list << new Vampire("Count Dracula", 1897)
```

5.2.2 Scala Lists

In Scala, you create a list and add to a list in a slightly different way:

```
var list = List[Vampire]();
list :+ new  Vampire("Count Dracula", 1897)
```

Also, this actually creates a new list, instead of modifying the existing list. This is because the default `List` in Scala is *immutable*, meaning it cannot be modified. Although this may seem strange in conjunction with *functional* programming, it makes parallel programming (programming for multiple processors) easier.

5.2.3 JavaScript Arrays

As mentioned earlier, JavaScript uses `Array`[1] instead of `List`.

[1] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array.

Arrays can be created much like lists in Groovy. However, the methods available are somewhat different. For example, push is used instead of add.

```
def array = []
array.push(new Vampire("Count Dracula", 1897))
```

You can also declare the initial values of Array. For example, the following two lines are equivalent:

```
def years = [1666, 1680, 1722]
def years = new Array(1666, 1680, 1722)
```

To add to the confusion, arrays in JavaScript can be accessed much like Java arrays. For example:

```
def firstYear = years[0]
def size = years.length
```

5.3 Sets

Set is much like List, but each value or object can only have one instance in Set.

Set has many of the same methods as List. In particular, it is missing the methods that use an index, because Set is not necessarily in any particular order.

```
Set<String> dragons = new HashSet<>();
dragons.add("Lambton");
dragons.add("Deerhurst");
dragons.size(); // 2
dragons.remove("Lambton");
dragons.size(); // 1
```

In Java, there is such a thing as SortedSet, which is implemented by TreeSet. For example, let's say you wanted a sorted list of names, as follows:

```
SortedSet<String> dragons = new TreeSet<>();
dragons.add("Lambton");
dragons.add("Smaug");
dragons.add("Deerhurst");
dragons.add("Norbert");
System.out.println(dragons);
// [Deerhurst, Lambton, Norbert, Smaug]
```

TreeSet will magically always be sorted in the proper order.

Okay, it's not really magic. The object to be sorted must implement the *comparable* interface, but you haven't learned about interfaces yet.

JavaScript does not yet have a built-in Set class.

5.4 Maps

Map is a collection of keys associated with values. It may be easier to understand with an example.

```
Map<String,String> map = new  HashMap<>();
map.put("Smaug", "deadly");
map.put("Norbert", "cute");
map.size(); // 2
map.get("Smaug"); // deadly
```

Map also has the following methods:

- containsKey(Object key): Returns true, if this map contains a mapping for the specified key
- containsValue(Object value): Returns true, if this map maps one or more keys to the specified value
- keySet(): Returns a Set view of the keys contained in this map
- putAll(Map m): Copies all of the mappings from the specified map to this map
- remove(Object key): Removes the mapping for a key from this map, if it is present

5.4.1 Groovy Maps

Just as for List, Groovy has a simpler syntax for creating and editing Map.

```
def map = ["Smaug": "deadly"]
map.Norbert = "cute"
println(map) // [Smaug:deadly, Norbert:cute]
```

5.4.2 Scala Maps

Scala's Map syntax is also somewhat shorter.

```
var map = Map("Smaug" -> "deadly")
var  map2 =  map + ("Norbert" -> "cute")
println(map2) // Map(Smaug -> deadly, Norbert -> cute)
```

As with List, Scala's default Map is also immutable.

5.4.3 JavaScript Maps

JavaScript does not yet have a built-in Map class, but it can be approximated by using the built-in Object[2] syntax. For example:

```
def map = {"Smaug": "deadly", "Norbert": "cute"}
```

You could then use either of the following to access map values: `map.Smaug` or `map["Smaug"]`.

5.5 Summary

This chapter introduced you to the following concepts:

- *Arrays*: Primitive collections of data
- *Lists*: An expandable collection of objects or values
- *Sets*: A collection of unique objects or values
- *Maps*: A dictionary-like collection

[2]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object.

CHAPTER 6

Conditionals and Loops

To rise above the label of *calculator*, a programming language must have conditional statements and loops.

A *conditional statement* is a statement that may or may not execute, depending on the circumstances.

A *loop* is a statement that gets repeated multiple times.

6.1 If, Then, Else

The most basic conditional statement is the *if* statement. It is the same in all languages covered in this book. For example:

```
if (vampire) { // vampire is a boolean
        useWoodenStake();
}
```

Curly brackets ({}) define a block of code (in Java, Scala, Groovy, and JavaScript). To define what should happen if your condition is `false`, you use the `else` keyword.

```
if (vampire) {
        useWoodenStake();
} else {
        useAxe();
}
```

Actually, this can be shortened, because we only have one statement per condition.

```
if (vampire) useWoodenStake();
else useAxe();
```

But it's generally better to use the curly-bracket style in Java. If you have multiple conditions you have to test for, you can use the `else if` style, such as the following:

```
if  (vampire) useWoodenStake();
else if (zombie) useBat();
else useAxe();
```

A. L. Davis, *Modern Programming Made Easy*, DOI 10.1007/978-1-4842-2490-8_6

6.2 switch Statements

Sometimes you have so many conditions that your `else if` statements span several pages. In this case, you might consider using the `switch` keyword. It allows you to test for several different values of the same variable. For example:

```
switch (monsterType) {
case "Vampire": useWoodenStake(); break;
case "Zombie": useBat(); break;
case "Orc": shoutInsult();
default: useAxe();
}
```

The `case` keyword denotes the value to be matched.

The `break` keyword always causes the program to exit the current code block. This is necessary in a `switch` statement; otherwise, every statement after the `case` will be executed. For example, when `monsterType` is `"Orc"`, `shoutInsult` and `useAxe` are executed.

The `default` keyword denotes the code to execute if none of the cases is matched. It is much like the final `else` block of an `if/else` block.

There is more to `switch` statements, but this involves concepts I'll cover later on, so we'll return to this topic.

6.3 Boolean Logic

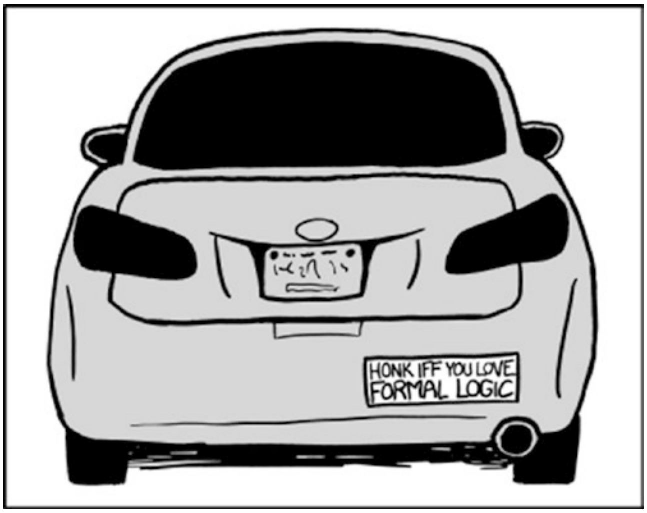

Figure 6-1. *Formal Logic—XKCD 1033 (courtesy* `http://xkcd.com/1033/`*)*

Computers use a special kind of math called *Boolean logic* (it's also called *Boolean algebra*). All you really need to know are the following three Boolean operators and six comparators:

&&—AND: True only if left and right values are true.

||—OR y if either left or right value is true.

!—NOT: Negates a Boolean (true becomes false; false becomes true.

==: Equals

!=: Does not equal

<: Less than

>: Greater than

<=: Less than or equal

>=: Greater than or equal

Conditions (such as if) operate on Boolean values (true/false)—the same boolean type that you learned about in Chapter 3. When used properly, all the preceding operators result in a Boolean value.

For example:

```
if (age > 120 && skin == Pale && !winkled) {
        probablyVampire();
}
```

6.4 Looping

The two simplest ways to loop are the while loop and do/while loop.

The while loop simply repeats until the *loop condition* becomes false.

```
boolean repeat = true;
while (repeat) {
        doSomething();
        repeat = false;
}
```

The preceding would call the doSomething() method once. The loop condition in the preceding code is repeat. This is a simple example. Usually, the loop condition would be something more complex.

The do loop is like the while loop, except that it always goes through at least one time. For example:

```
boolean repeat = false;
do  {
        doSomething();
} while(repeat);
```

It's often helpful to increment a number in your loop, for example:

```java
int i = 0;
while (i < 10) {
        doSomething(i);
        i++;
}
```

The preceding loop can be condensed using the `for` loop.

```java
for  (int  i = 0; i < 10; i++) {
        doSomething(i);
}
```

The `for` loop has an initiation clause, a loop condition, and an increment clause. This style of loop is useful for looping through an array with an index. For example:

```java
String[] strArray = {"a", "b", "c"};
for (int i = 0; i < strArray.length; i++)
        System.out.print(strArray[i]);
```

This would print "abc." The preceding loop is equivalent to the following:

```java
int i = 0;
while  (i < strArray.length) {
    String str = strArray[i];
        System.out.print(str);
        i++;
}
```

In Java, you can write for loops in a more concise way for an array or collection (list or set). For example:

```java
String[] strArray = {"a", "b", "c"};
for  (String str : strArray)
          System.out.print(str);
```

This is called a `for` each loop. Note that it uses a colon instead of a semicolon.

6.5 Summary

In this chapter, you learned about the following:

- Using the `if` statement
- How to use Boolean logic
- `switch` statements
- Using `for`, `do`, and `while` loops

CHAPTER 7

Methods

A *method* is a series of statements combined into one block inside a class and given a name. In the Cold War days, these were called sub-routines, and many other languages call them *functions*. However, the main difference between a method and a function is that a method has to be associated with a class, whereas a function does not.

7.1 Call Me

Methods exist to be called. You can think of a method as a message that is sent or a command given. To *call* a method (also known as *invoking* a method), you simply write the name of the object, a dot, then the method name. For example:

```
Dragon dragon = new  Dragon();
dragon.fly(); // dragon is the object, and fly is the method
```

The `fly` method would be defined within the `Dragon` class.

```
public void fly() {
        // flying code
}
```

Void In Java, `void` means that no result is returned.

Methods can also have parameters. A *parameter* is a value (or reference value) that is part of a method call. Together, the method's name and parameters are called the *method signature*. For example, the following method has two parameters:

```
public void fly(int x, int y) {
        // fly to that x, y coordinate.
}
```

7.1.1 Non-Java

Other languages define methods (or functions) differently. For example, in Groovy, you can use the def keyword to define a method (in addition to Java's normal syntax), as follows:

```
def fly() { println("flying") }
```

Scala also uses the def keyword to define a method, but you also need an equal (=) sign.

```
def fly() = { println("flying") }
```

JavaScript uses the function keyword to define a function:

```
function fly() { alert("flying") }
```

7.2 Break It Down

Methods also exist to organize your code. One rule of thumb is to never have a method that is longer than one screen. It makes no difference to the computer, but it makes all the difference to humans (including you).

It's also very important to name your method well. For example, a method that fires an arrow should be called "fireArrow," and not "fire," "arrow," or "arrowFartBigNow."

This may seem like a simple concept, but you might be surprised by how many people fail to grasp it.

7.3 Return to Sender

Often, you will want a method to return a result. In Java, you use the return keyword to do this. For example:

```
public Dragon makeDragonNamed(String name) {
return new Dragon(name);
}
```

Once the return statement is reached, the method is complete. Whatever code called the method will resume execution.

In some languages, such as Groovy and Scala, the return keyword is optional. Whatever value is put on the last line will get returned. For example (Groovy):

```
def makeDragonNamed(name) {
        new Dragon(name)
}
```

7.4 Static

In Java, a *static method* is a method that is not linked to an object instance. However, it must be part of a class.

For example, the `random()` method in the `java.util.Math` class we learned about earlier is a static method.

To declare a static method, you simply add the word `static`, as in the following:

```java
public static String makeThemFight(Dragon d, Vampire v) {
        // a bunch of code goes here.
}
```

Because Java is an object-oriented programming language (OOP), in theory, static methods should be used sparingly, because they are not linked to any object. However, in real life, you will see them *a lot*.

7.5 Varargs

Varargs, or "variable arguments," allow you to declare a method's last parameter with an ellipsis (`...`), and it will be interpreted to accept any number of parameters (including zero) and convert them into an array in your method. For example, see the following code:

```java
void printSpaced(Object... objects) {
        for (Object o : objects) System.out.print(o + " ");
}
```

Putting it all together, you can have the following code (with output in comments):

```java
printSpaced("A", "B", "C"); // A B C
printSpaced(1, 2, 3); // 1 2 3
```

7.6 Main Method

Now that you know about static methods, you can finally run a Java program (sorry it took so long). Here's how you create an executable *main method* in Java:

```java
import static java.lang.System.out;
/** Main class. */
public class Main {
    public static void main(String ... args) {
        out.println("Hello World!");
    }
}
```

Then, to compile it, open your command prompt or terminal and type the following:

```
javac Main.java
java Main
```

Or, in NetBeans, do the following:

- Right-click the `Main` class.
- Choose Run File.

7.7 Exercises

Try out methods After you've created the `Main` class, try adding some methods to it. Try calling methods from other methods and see what happens.

Lists, Sets, and Maps In Java, all of these data structures are under the `java.util` package. So, start by importing this whole package:

```
import    java.util.*;
```

Then go back to Chapter 5 and try out some of the code there.

7.8 Summary

This chapter explained the concept of methods and how they should be used.

We also put together everything you've learned up to this point and made a small Java application.

PART III

Polymorphic Spree

The title of this part is a play on the name of the band Polyphonic Spree.[1]

The term *polymorphism* refers to a principle in biology according to which an organism or species can have many different forms or stages. This principle can also be applied to object-oriented programming (OOP) languages such as Java. Subclasses of a class can define their own unique behaviors and yet share some of the same functionality of the parent class.

Figure 1. *Mythical creatures (clockwise from bottom-left): unicorn, griffon, phoenix, dragon, roc (center)*

[1]`www.thepolyphonicspree.com/`.

CHAPTER 8

Inheritance

Inheritance is a good way to share functionality between objects. When a class has a parent class, we say it *inherits* the fields and methods of its parent.

In Java, you use the `extends` keyword to define the parent of a class. For example:

```
public class Griffon extends FlyingCreature {
}
```

Another way to share functionality is called *composition*. This means that an object holds a reference to another object and uses it to do things. For example:

```
class Griffon {
    Wing leftWing = new Wing()
    Wing rightWing = new Wing()
    def fly() {
        leftWing.flap()
        rightWing.flap()
    }
}
```

This way, you can also have a `Bird` class that also uses the `Wing` class, for example.

8.1 Objectify

What is an object anyway? An *object* is an instance of a class (in Java, Groovy, and Scala).

In Java, classes have constructors, which can have multiple parameters for initializing the object. For example, see the following:

```
class  FlyingCreature  {
        String name;
        // constructor
        public  FlyingCreature(String name) {
            this.name = name;
        }
}
```

A. L. Davis, *Modern Programming Made Easy*, DOI 10.1007/978-1-4842-2490-8_8

The constructor of FlyingCreature has one parameter, name, which is stored in the name field. A constructor must be called using the new keyword, to create an object, for example:

```
String name = "Bob";
FlyingCreature fc = new  FlyingCreature(name);
```

Once an object is created, it can be passed around (this is called a *pass by reference*). Although String is a special class, it is a class, so you can pass around an instance of it, as shown in the preceding code.

8.1.1 JavaScript

In JavaScript, a constructor is a function used to define a prototype. Inside the constructor, the prototype is referred to using the this keyword. For example, you could define a Creature in JavaScript, as follows:

```
function Creature(n) {
    this.name = n;
}
var  bob = new  Creature('Bob');
```

■ **Note** All functions and objects in JavaScript have a prototype.

8.2 Parenting 101

A *parent class* defines shared functionality (methods) and state (fields) that are common to multiple classes.

For example, let's create a FlyingCreature class that defines a fly() method and has a name.

```
class FlyingCreature {
        String name;
        public FlyingCreature(String name) {
                this.name = name;
        }
        public void fly() {
                System.out.println(name + " is flying");
        }
}
class Griffon extends FlyingCreature {
        public  Griffon(String n) { super(n); }
}
class Dragon extends FlyingCreature {
        public  Dragon(String n) { super(n); }
}
public  class  Parenting  {
        public static void main(String ... args) {
                Dragon d = new  Dragon("Smaug");
                Griffon g = new   Griffon("Gilda");
                d.fly(); // Smaug is flying
                g.fly(); // Gilda is flying
        }
}
```

There are two classes in the preceding code, Griffon and Dragon, that extend FlyingCreature. FlyingCreature is sometimes referred to as the *base class*. Griffon and Dragon are referred to as *subclasses*.

Keep in mind that you can use the parent class's type to refer to any subclass. For example, you can make any flying creature fly, as follows:

```
FlyingCreature creature = new Dragon("Smaug");
creature.fly(); // Smaug is flying
```

This concept is called *extension*. You *extend* the parent class (FlyingCreature, in this case).

8.2.1 JavaScript

In JavaScript, we can use prototypes to extend functionality.

For example, let's say we have a prototype called Undead.

```
function Undead() {
    this.dead = false;
    this.living = false;
}
```

Now let's create two other constructors, Zombie and Vampire.

```
function Zombie() {
    Undead.call(this);
    this.deseased = true;
    this.talk = function() { alert("BRAINS!") }
}
Zombie.prototype = Object.create(Undead.prototype);

function Vampire() {
    Undead.call(this);
    this.pale = true;
    this.talk = function() { alert("BLOOD!") }
}
Vampire.prototype = Object.create(Undead.prototype);
```

Note how we set `Zombie`'s and `Vampire`'s prototype to an instance of the `Undead` prototype. This allows zombies and vampires to inherit the properties of `Undead`, while having different `talk` functions, as follows:

```
var zombie = new Zombie();
var vamp = new Vampire();
zombie.talk();   //BRAINS
zombie.deseased;  // true
vamp.talk();     //BLOOD
vamp.pale; //true
vamp.dead; //false
```

8.3 Packages

In Java (and related languages, Groovy, and Scala), a *package* is a namespace for classes. *Namespace* is a just shorthand for a bin of names. Every modern programming language has some type of namespace feature. This is necessary, owing to the nature of having lots and lots of classes in typical projects.

As you learned in Chapter 3, the first line of a Java file defines the package of the class, for example:

```
package com.github.modernprog;
```

Also, there is a common understanding that a package name corresponds to a URL (`github.com/modernprog`, in this case). However, this is not necessary.

8.4 Public Parts

You might be wondering why the word *public* shows up everywhere in the examples so far. The reason has to do with encapsulation. *Encapsulation* is a big word that just means "a class should expose as little as possible to get the job done" (some things are meant to be private). This helps reduce complexity of code and, therefore, makes it easier to understand and think about.

There are three different keywords in Java for varying levels of "exposure."

- `Private`: Only this class can see it.
- `Protected`: Only this class and its descendants can see it.
- `Public`: Everyone can see it.

There's also "default" protection (absent of a keyword), which limits use to any class in the same package.

This is why classes tend to be declared `public`, because, otherwise, their usage would be very limited. However, a class can be private, for example, when declaring a class within another class, as follows:

```
public class Griffon extends FlyingCreature {
        private class GriffonWing {}
}
```

8.4.1 JavaScript

JavaScript does not have the concept of packages, but, instead, you must rely on `scope`. Variables are only visible inside the function they were created in, except for *global* variables.

8.5 Interfaces

An *interface* declares method signatures that will be implemented by classes that extend the interface. This allows Java code to work on several different classes without necessarily knowing what specific class is "underneath" the interface.

For example, you could have an interface with one method, as follows:

```
public interface  Beast  {
        int getNumberOfLegs(); // all interface methods are public
}
```

Then you could have several different classes that *implement* that interface.

```
public class Griffon extends FlyingCreature implements  Beast {
         public int getNumberOfLegs() { return 2; }
}
public class Unicorn implements Beast {
         public int getNumberOfLegs() { return 4; }
}
```

■ **Note** JavaScript does not have an equivalent concept to interface.

8.6 Abstract Class

An *abstract* is a class that can have abstract methods but cannot have instances. It is something like an interface with functionality, however, a class can only extend one superclass, while it can implement multiple interfaces.

For example, to implement the preceding Beast interface as an abstract class, you can do the following:

```
public abstract class Beast {
        public abstract int getNumberOfLegs();
}
```

Then you could add non-abstract methods and/or fields.

8.7 Enums

In Java, the `enum` keyword creates a type-safe, ordered list of values. For example:

```
public enum BloodType {
        A, B, AB, O, VAMPIRE, UNICORN;
}
```

An `enum` variable can only point to one of the values in the enum. For example:

```
BloodType type = BloodType.A;
```

The enum is automatically given a bunch of methods, such as

- `values()`: Gives you an array of all possible values in the enum (static)
- `valueOf(String)`: Converts the given string into the enum value with the given name
- `name()`: An instance method on the enum that gives its name

Also, enums have special treatment in `switch` statements. For example, in Java, you can use an abbreviated syntax (assuming `type` is a `BloodType`).

```
switch (type) {
        case VAMPIRE: return vampire();
        case UNICORN: return unicorn();
        default: return human();
}
```

8.8 Annotations

Java annotations allow you to add meta-information to Java code that can be used by the compiler, various APIs, or even your own code at runtime.

The most common annotation you will see is the `@Override` annotation, which declares to the compiler that you are overriding a method. For example:

```java
@Override
public String toString() {
        return "my own string";
}
```

This is useful, because it will cause a compile-time error if you mistype the method name, for example.

Other useful annotations are those in `javax.annotation`, such as `@Nonnull` and `@Nonnegative`, which declare your intentions.

Annotations such as `@Autowired` and `@Inject` are used by direct-injection frameworks such as Spring and Google Guice,[1] repectively, to reduce “wiring” code.

8.9 Autoboxing

Although Java is an object-oriented language, this sometimes conflicts with its primitive types (`int, long`, `float`, `double`, etc.). For this reason, Java added autoboxing and unboxing to the language.

8.9.1 Autoboxing

The Java compiler will automatically wrap a primitive type in the corresponding object when it’s necessary, for example, when passing in parameters to a function or assigning a variable, as in the following: `Integer number = 1`.

8.9.2 Unboxing

This is simply the reverse of autoboxing. The Java compiler will unwrap an object to the corresponding primitive type, when possible. For example, the following code would work: `double d = new Double(1.1) + new Double(2.2)`.

8.10 Summary

After reading this chapter, you should understand OOP, polymorphism, and the definitions of the following:

- Extension and composition
- Public vs. private vs. protected
- Class, abstract class, interface, enum
- Annotations
- Autoboxing and unboxing

[1]http://code.google.com/p/google-guice/.