CHAPTER 9

Design Patterns

In object-oriented programming (OOP), design patterns are useful organizations of state and behavior that make your code more readable, testable, and extensible.

9.1 Observer

The *observer* pattern allows you to broadcast information from one class to many others, without them having to know about each other directly.

It is often used with events. For example, the `KeyListener`, `MouseListener`, and many other "Listener" interfaces in Java Swing implement the observer pattern and use events.

Another example of this pattern is the `Observable` class and `Observer` interfaces supplied in Java. Here is a simple example that simply repeats the same event forever:

```
1   import       java.util.Observable;
2
3   public class EventSource extends Observable implements Runnable {
4       @Override
5       public void run() {
6           while  (true) {
7               notifyObservers("event");
8           }
9       }
10  }
```

Although the event is simply a string in this example, it could be of any type.

The following class implements the `Observer` interface and prints out any events of type `String`:

```
1   import java.util.Observable;
2   import java.util.Observer;    /* this is Event Handler */
3
```


A. L. Davis, *Modern Programming Made Easy*, DOI 10.1007/978-1-4842-2490-8_9

```
 4   public class StringObserver implements Observer {
 5       public void update(Observable obj, Object event) {
 6           if (event instanceof String) {
 7               System.out.println("\nReceived Response: " + event );
 8           }
 9       }
10   }
```

To run this example, simply do the following:

```
1   final EventSource eventSource = new EventSource();
2   // create an observer
3   final StringObserver stringObserver = new StringObserver();
4   // subscribe the observer to the event source
5   eventSource.addObserver(stringObserver);
6   // starts the event thread
7   Thread thread = new  Thread(eventSource);
8   thread.start();
```

9.2 MVC

Model–view–controller (MVC) is possibly the most popular software design pattern (Figure 9-1). As the name suggests, it consists of three major parts:

- *Model*: The data or information being shown and manipulated
- *View*: What actually defines how the model is shown to the user
- *Controller*: Defines how actions can manipulate the model

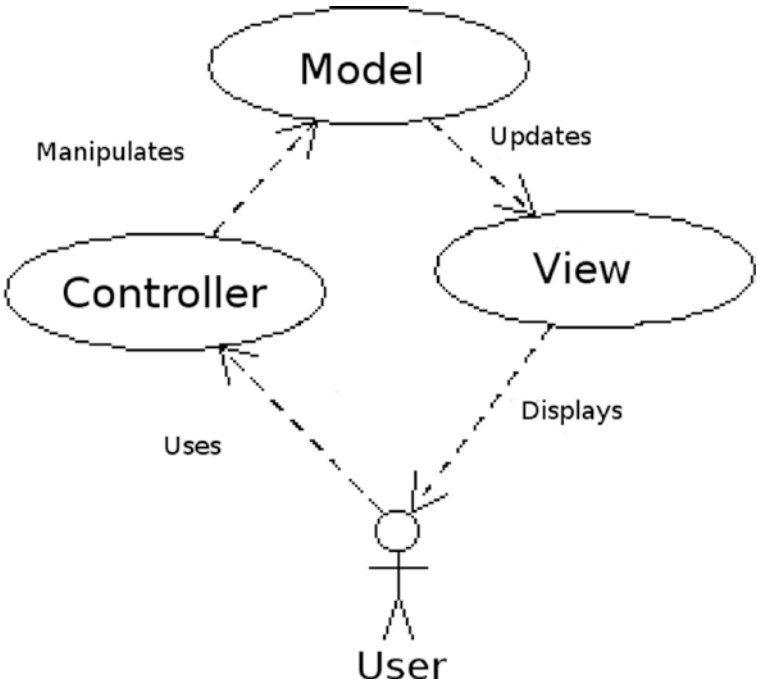

Figure 9-1. *Model–view–controller*

This design allows the controller, model, and view to know very little about each other. This reduces *coupling*—the degree to which different components of the software rely on other components. When you have low coupling, your software is easier to understand and easier to extend.

We will look at a great example of MVC when I talk about grails.

9.3 DSL

A *domain specific language* (DSL) is a custom programming language made for a specific domain. For example, you can think of HTML as a DSL for displaying web pages.

Some languages allow you such freedom that you can create a DSL inside the language. For example, Groovy and Scala allow you to override the math symbols (+, -, etc.). The other freedoms of these languages (optional parentheses and semicolons) allow for DSL-like interfaces. We call these DSL-like interfaces *fluent interfaces*.

You can also create fluent interfaces in Java and other languages.

9.3.1 Closures

Within Groovy, you can take a block of code (a closure) as a parameter and then call it, using a local variable as a delegate. For example, imagine that you have the following code for sending SMS texts:

```
class SMS {
        def from(String fromNumber) {
                // set the from
        }
        def to(String toNumber) {
                // set the to
        }
        def body(String body) {
                // set the body of text
        }
        def send() {
                // send the text.
        }
}
```

In Java, you'd have to use this the following way:

```
SMS m = new  SMS();
m.from("555-432-1234");
m.to("555-678-4321");
m.body("Hey there!");
m.send();
```

In Groovy, you can add the following `static` method to the SMS class for DSL-like usage:

```
def static send(block) {
        SMS m = new SMS()
        block.delegate = m
        block()
        m.send()
}
```

This sets the SMS object as a delegate for the block, so that methods are forwarded to it. With this you can now do the following:

```
SMS.send {
        from '555-432-1234'
        to '555-678-4321'
        body 'Hey there!'
}
```

9.3.2 Overriding Operators

In Scala or Groovy, you could create a DSL for calculating speeds with specific units, such as meters per second.

```
val time = 20 seconds
val dist = 155 meters
val speed = dist / time
println(speed.value) //  7.75
```

By overriding operators, you can constrain users of your DSL to reduce errors. For example, `time/dist` would cause a compilation error in this DSL.

Here's how you would define this DSL in Scala:

```
class Second(val value: Float) {}
class MeterPerSecond(val  value:  Float) {}
class Meter(val value: Float) {
  def /(sec: Second) = {
    new MeterPerSecond(value / sec.value)
  }
}
class EnhancedFloat(value: Float) {
  def seconds =  {
    new   Second(value)
  }
  def  meters =  {
    new  Meter(value)
  }
}
implicit  def  enhanceFloat(f:  Float) =  new  EnhancedFloat(f)
```

 Scala has the `implicit` keyword, which allows the compiler to do implicit conversions for you.

Notice how the divide / operator is defined just like any other method.

 In Groovy, you overload operators by defining methods with special names[1] such as `plus`, `minus`, `multiply`, `div`, etc.

---

[1] http://groovy.codehaus.org/Operator+Overloading.

9.4 Actors

The *actor design pattern* is a useful pattern for developing concurrent software. In this pattern, each actor executes in its own thread and manipulates its own data. The data cannot be manipulated by anyone else. Messages are passed between actors to cause them to change data (Figure 9-2).

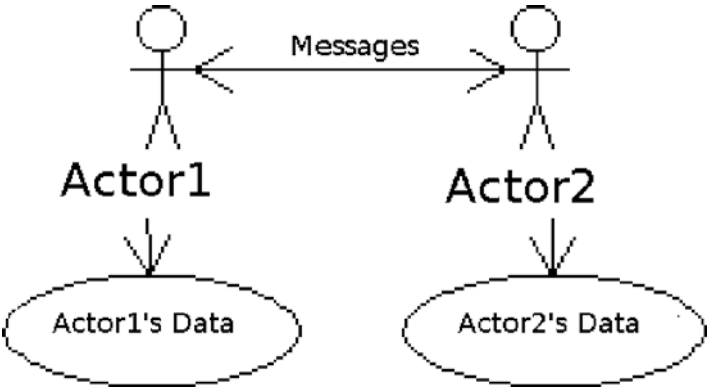

Figure 9-2. *Actors*

■ **Note** When data can only be changed by one thread at a time, we call it *thread-safe*.

There are many implementations of this pattern that you can use, including the following:

- Akka[2]
- Jetlang[3]
- FunctionalJava[4]
- GPars[5]

[2] http://akka.io/.
[3] https://code.google.com/p/jetlang/.
[4] http://functionaljava.org/.
[5] http://gpars.codehaus.org/.

CHAPTER 10

Functional Programming

Functional programming (FP) is a programming style that focuses on functions and minimizes changes of state (using immutable data structures). It is closer to expressing solutions mathematically, rather than through step-by-step instructions.

In FP, functions should be "side-effect free" (nothing outside the function is changed) and *referentially transparent* (a function returns the same value every time when given the same arguments).

FP is an alternative to the more common *imperative programming*, which is closer to telling the computer the steps to follow.

Although functional-programming could be achieved in Java pre-Java-8,[1] Java 8 enabled language-level FP support with lambda expressions and *functional interfaces*.

Java 8, JavaScript, Groovy, and Scala all support functional-style programming, although they are not FP languages.

Note Prominent functional programming languages such as Common Lisp, Scheme, Clojure, Racket, Erlang, OCaml, Haskell, and F# have been used in industrial and commercial applications by a wide variety of organizations. Clojure[2] is a Lisp-like language that runs on the JVM.

10.1 Functions and Closures

Of course, "functions as a first-class feature" is the basis of functional programming. *First-class feature* means that a function can be used anywhere a value can be used.

For example, in JavaScript, you can assign a function to a variable and call it

```
1   var func = function(x) { return x + 1; }
2   var three = func(2); //3
```

[1] http://functionaljava.org/.
[2] http://clojure.org/.

A. L. Davis, *Modern Programming Made Easy*, DOI 10.1007/978-1-4842-2490-8_10

Although Groovy doesn't have first-class functions, it has something very similar: closures. A closure is simply a block of code wrapped in curly brackets with parameters defined left of the -> (arrow). For example:

```
1   def closr = {x -> x + 1}
2   println( closr(2) ); //3
```

If a closure has one argument, it can be referenced as `it` in Groovy. For example:

```
1   def closr = {it + 1}
```

Java 8 introduced the lambda expression, which is something like a closure that implements an interface. The main syntax of a lambda expression is "parameters -> body." The Java compiler uses the context of the expression to determine which functional interface is being used (and the types of the parameters). For example:

```
1   Function<Integer,Integer> func = x -> x + 1;
2   int three = func.apply(2);
```

Here, the functional interface is `Function`, which has the `apply` method. The return value and parameter type of both `Integers`, thus `Integer,Integer`, are the generic type parameters.

 In Java 8, a *functional interface* is defined as an interface with exactly one abstract method. This even applies to interfaces that were created with previous versions of Java.

In Scala, everything is an expression, and functions are a first-class feature. Here's a function example in Scala:

```
1   var  f =  (x:  Int) =>  x + 1;
2   println(f(2));
```

Although both Java and Scala are statically typed, Scala actually uses the right-hand side to infer the type of function being declared, whereas Java does the opposite.

 In Java, Groovy, and Scala, the `return` keyword can be omitted, if there is one expression in the function/closure. However, in Groovy and Scala, the `return` keyword can also be omitted, if the returned value is the last expression.

10.2 Map/Filter/etc.

Once you have mastered functions, you quickly realize that you need a way to perform operations on collections (or sequences or streams) of data.

Because these are common operations, *sequence operations,* such as `map`, `filter`, `reduce`, etc., were invented.

We'll use JavaScript for the examples in this section, because it is easier to read, and the function names are fairly standard across programming languages.

`map` translates or changes input elements into something else (Figure 10-1).

```
1   var names = persons.map(function(person) { return person.name })
```

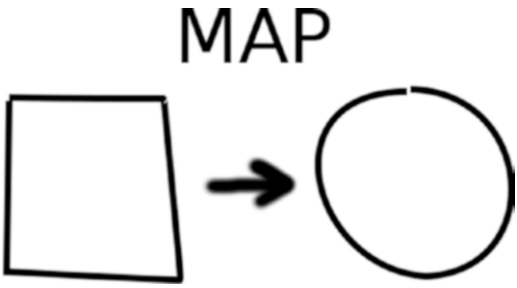

Figure 10-1. *Map*

`filter` gives you a subset of elements (what returns `true` from some *predicate* function [Figure 10-2]).

```
1   var adults = persons.filter(function(person) { return person.age >= 18 })
```

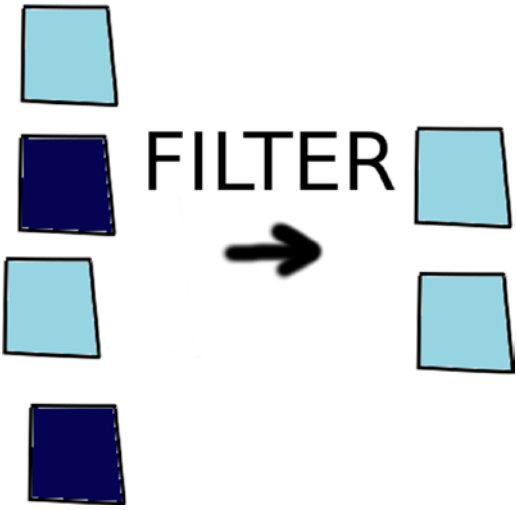

Figure 10-2. *Filter*

`reduce` performs a reduction on the elements (Figure 10-3).

```
1   var totalAge = persons.reduce(function(total, p) { return total+p.age }, 0)
```

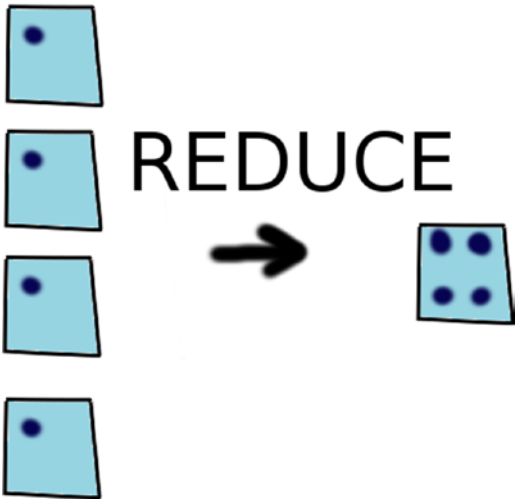

Figure 10-3. *Reduce*

`limit` gives you only the first N elements (Figure 10-4).

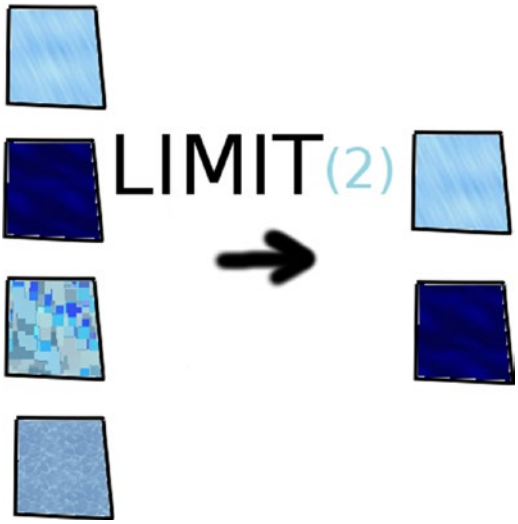

Figure 10-4. *Limit*

`concat` combines two different collections of elements (Figure 10-5).

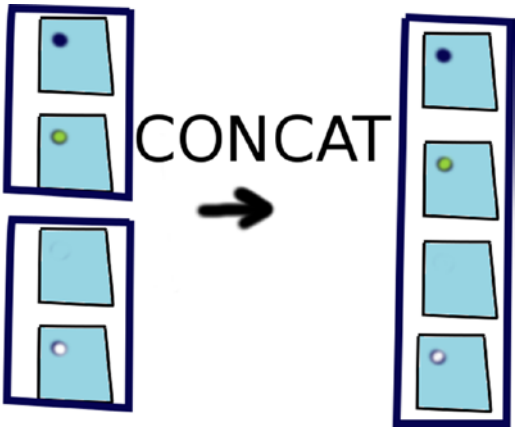

Figure 10-5. *Concat*

10.3 Immutability

Immutability and FP go together like peanut butter and jelly. Although it's not necessary, they blend nicely.

In purely functional languages, the idea is that each function has no effect outside itself—no side effects. This means that every time you call a function, it returns the same value given the same inputs.

To accommodate this behavior, there are *immutable* data-structures. An immutable data-structure cannot be directly changed but returns a new data-structure with every operation.

For example, as you learned earlier, Scala's default `Map` is immutable.

```
1   val map = Map("Smaug" -> "deadly")
2   val map2 = map + ("Norbert" -> "cute")
3   println(map2) // Map(Smaug -> deadly, Norbert -> cute)
```

So, in the preceding, `map` would remain unchanged.

Each language has a keyword for defining immutable variables (values).

- Scala uses the `val` keyword to denote immutable values, as opposed to `var`, which is used for mutable variables.
- Java has the `final` keyword for declaring immutable variables.
- In addition to the `final` keyword, Groovy includes the `@Immutable` annotation[3] for declaring a whole class immutable.
- JavaScript uses the `const` keyword.[4]

[3] http://groovy.codehaus.org/Immutable+AST+Macro.

[4] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Values,_variables,_and_literals#Constants.

For example (in Groovy):

```
1   public class Centaur {
2       final String name
3       public Centaur(name) {this.name=name}
4   }
5   Centaur c = new Centaur("Bane");
6   println(c.name) // Bane
7
8   c.name = "Firenze" //error
```

This works for simple references and primitives, such as numbers and strings, but for things such as lists and maps, it's more complicated. For these cases, open source immutable libraries have been developed for the languages in which it's not included, such as the following:

- Guava[5] for Java and Groovy
- Immutable-JS[6] for JavaScript

10.4 Java 8

In Java 8, the *Stream* interface was introduced. A stream is like an improved iterator that supports chaining methods to perform complex operations.

To use a stream, you must first create one in one of the following ways:

- `Collection's stream()` *method or* `parallelStream()` *method*: These create a stream backed by the collection.
- `Arrays.stream()` *method*: Used for converting arrays to streams
- `Stream.generate(Supplier<T> s)`: Returns an infinite sequential stream in which each element is generated by the given supplier
- `Stream.iterate(T seed, UnaryOperator<T> f)`: Returns an infinite sequential ordered stream produced by iterative application of a function to an initial element seed, producing a stream consisting of seed, f(seed), f(f(seed)), etc.

Once you have a stream, you can then use `filter`, `map`, and `reduce` operations to concisely perform calculations on the data. For example:

```
1   String longestName = dragons.stream()
2       .filter(d -> d.name != null)
3       .map(d -> d.name)
4       .reduce((n1, n2) -> n1.length() > n2.length() ? n1 : n2)
5       .get();
```

[5] https://code.google.com/p/guava-libraries/.
[6] https://github.com/facebook/immutable-js.

10.5 Groovy

In Groovy, `findAll` and other methods are available on every object but are especially useful for lists and sets. The following method names are used in Groovy:

- `findAll`: Much like `filter`, it finds all elements that match a closure.
- `collect`: Much like `map`, this is an iterator that builds a collection.
- `inject`: Much like `reduce`, it loops through the values and returns a single value.
- `each`: Iterates through the values using the given closure
- `eachWithIndex`: Iterates through with two parameters: a value and an index
- `find`: Finds the first element that matches a closure
- `findIndexOf`: Finds the first element that matches a closure and returns its index
- `any`: `True` if any element returns `true` for the closure
- `every`: `True` if all elements return `true` for the closure

For example, the following assumes `dragons` is a list of dragon objects:

```
1   String longestName = dragons.
2   findAll { it.name != null }.
3   collect { it.name }.
4   inject("") { n1, n2 -> n1.length() > n2.length() ? n1 : n2 }
```


Remember that `it` in Groovy can be used to reference the single argument of a closure.

10.6 Scala

Scala has many such methods on its built-in collections, including the following:

- `map`: Converts values
- `flatMap`: Converts values and then concatenates the results together
- `filter`: Limits the returned values, based on some Boolean expression
- `find`: Returns the first value matching the given predicate

- `forAll`: `True` only if all elements match the given predicate
- `exists`: `True` if at least one element matches the given predicate
- `foldLeft`: Reduces the values to one value using the given closure, starting at the last element and going left
- `foldRight`: Same as `foldLeft`, but starting from the first value and going up

For example, you can use `map` to perform an operation on a list of values, as follows:

```
1   val list = List(1, 2, 3)
2   list.map(_  * 2) // List(2, 4, 6)
```

 Much like `it` in Groovy, in Scala, you can use the underscore to reference a single argument.

Assuming `dragons` is a List of dragon objects, you can do the following in Scala:

```
1   var longestName = dragons.filter(_ != null).map(_.name).foldRight("")(
2       (n1:String,n2:String) => if (n1.length() > n2.length()) n1 else n2)
```

10.7 Summary

In this chapter, you should have learned about

- Functions as a first-class feature
- Map/Filter/Reduce
- Immutability and how it relates to FP
- Various features supporting FPs in Java, Groovy, Scala, and JavaScript

CHAPTER 11

Refactoring

Refactoring[1] means changing code in a way that has no effect on functionality. It is only meant to make the code easier to understand or to prepare for some future addition of functionality For example, sometimes you refactor code to make it easier to test.

There are two categories of refactoring I am going to cover, Object-Oriented and Functional, corresponding to the two different programming styles.

11.1 Object-Oriented Refactoring

The following actions are common refactorings in OOP:

- Changing a method or class name (renaming)
- Moving a method from one class to another (delegation)
- Moving a field from one class to another
- Creating a new class using a set of methods and fields from a class
- Changing a local variable to a class field
- Replacing a bunch of literals (strings or numbers) with a constant (static final)
- Moving a class from an anonymous class to a top level class
- Renaming a field

11.2 Functional Refactoring

The following actions are common refactorings in FP:

- Renaming a function
- Wrapping a function in another function and calling it

[1]Yes *refactoring* is a word!

A. L. Davis, *Modern Programming Made Easy*, DOI 10.1007/978-1-4842-2490-8_11

- Inline a function wherever it is called
- Extract common code into a function (the opposite of the previous.
- Renaming a function parameter
- Adding a parameter

You might notice some similarities between both lists. The principles of refactoring are universal.

11.3 Refactoring Examples

Here are some examples of refactoring code:

11.3.1 Renaming a Method

Before:

```
1   public static void main(String...args) {
2       animateDead();
3   }
4   public  static void  animateDead() {}
```

After:

```
1   public static void main(String...args) {
2       doCoolThing();
3   }
4   public  static void  doCoolThing() {}
```

11.3.2 Moving a Method from One Class to Another (Delegation)

Before:

```
1   public static void main(String...args) {
2       animateDead();
3   }
4   public  static void  animateDead() {}
```

After:

```
1   public class Animator() {
2       public void animateDead() {}
3   }
4   public static void main(String...args) {
5       new Animator().animateDead();
6   }
```

11.3.3 Replacing a Bunch of Literals (Strings or Numbers) with a Constant (Static Final)

Before:

```
1   public static void main(String...args) {
2       animateDead(123);
3       System.out.println(123);
4   }
5   public static void animateDead(int n) {}
```

After:

```
1   public static final int NUM = 123;
2   public static void main(String...args) {
3       animateDead(NUM);
4       System.out.println(NUM);
5   }
6   public static void animateDead(int n) {}
```

11.3.4 Renaming a Function

Before:

```
1   function castaNastySpell() { /* cast a spell here */ }
```

After:

```
1   function castSpell() { /* cast a  spell here  */ }
```

11.3.5 Wrapping a Function in Another Function and Calling It

Before:

```
1   castSpell('my cool spell');
```

After:

```
1   (function(spell) { castSpell(spell) })('my cool spell');
```

11.3.6 Inline a Function Wherever It Is Called

Before:

```
1   function castSpell(spell) { alert('You cast ' + spell); }
2   castSpell('crucio');
3   castSpell('expelliarmus');
```

After:

```
1   alert('You cast ' + 'crucio');
2   alert('You cast ' + 'expelliarmus');
```

11.3.7 Extract Common Code into a Function (the Opposite of the Previous)

Before:

```
1   alert('You cast crucio');
2   alert('You cast expelliarmus');
```

After:

```
1   function castSpell(spell) { alert('You cast ' + spell); }
2   castSpell('crucio');
3   castSpell('expelliarmus');
```

CHAPTER 12

Utilities

The `java.util` package contains many useful classes for everyday programming. Likewise, JavaScript and other languages come with many built-in objects for doing common tasks. I am going to cover a few of these.

12.1 Dates and Times

You should never store date values as text. It's too easy to mess up.

A. L. Davis, *Modern Programming Made Easy*, DOI 10.1007/978-1-4842-2490-8_12

12.1.1 Java 8 Date-Time

Java 8 comes with a new and improved Date-Time application program interface (API) that is much safer, easier to read, and more comprehensive than the previous API.

For example, creating a date looks like the following:

```
1   LocalDate date = LocalDate.of(2014, Month.MARCH, 2);
```

There's also a `LocalDateTime` class to represent date and time, `LocalTime` to represent only time, and `ZonedDateTime` to represent a time with a time zone.

Before Java 8, there were only two built-in classes to help with dates: `Date` and `Calendar`. These should be avoided.

- `Date` actually represents both a date and time.
- `Calendar` is used to manipulate dates.

In Java 7, you'd have to do the following to add five days to a date:

```
1   Calendar cal = Calendar.getInstance();
2   cal.setTime(date);
3   cal.add(5, Calendar.DAY);
```

12.1.2 Groovy Date

Groovy has a bunch of built-in features that make dates easier to work with. For example, numbers can be used to add/subtract days, as follows:

```
1   def date = new Date() + 5; //adds 5 days
```

Groovy also has TimeCategory[1] for manipulating dates and times. This lets you add and subtract any arbitrary length of time. For example:

```
1   import groovy.time.TimeCategory
2   now = new Date()
3   println now
4   use(TimeCategory) {
5       nextWeekPlusTenHours = now + 1.week + 10.hours - 30.seconds
6   }
7   println nextWeekPlusTenHours
```

A *Category* is a class that can be used to add functionality to other existing classes. In this case, TimeCategory adds a bunch of methods to the `Integer` class.

[1] http://groovy.codehaus.org/api/groovy/time/TimeCategory.html.

CATEGORIES

This is one of the many *meta-programming* techniques available in Groovy. To make a category, you create a bunch of static methods that operate on one parameter of a particular type (e.g., `Integer`). When the category is used, that type appears to have those methods. The object on which the method is called is used as the parameter. Take a look at the documentation for TimeCategory for an example of this in action.

12.1.3 JavaScript Date

JavaScript also has a `Date`[2] object.

You can create an instance of a `Date` object in several ways (these all create the same date):

```
1   Date.parse('June 13, 2014')
2   new Date('2014-06-13')
3   new Date(2014, 5, 13)
```

Note that if you adhere to the international standard (yyyy-MM-dd), a UTC time zone will be assumed; otherwise, it will assume you want a local time.

As usual with JavaScript, the browsers all have slightly different rules, so you have to be careful with this.

Don't ever use getYear In both Java and JavaScript, the `Date` object's `getYear()` method doesn't do what you think and should be avoided. For historical reasons, `getYear` does not actually return the year (e.g., 2014). You should use `getFullYear()` in JavaScript and `LocalDate` or `LocalDateTime` in Java 8.

12.1.4 Java DateFormat

Although `DateFormat` is in `java.text`, it goes hand-in-hand with `java.util.Date`.

The `SimpleDateFormat` is useful for formatting dates in any format you want. For example:

```
1   SimpleDateFormat sdf = new  SimpleDateFormat("MM/dd/yyyy");
2   Date date = new  Date();
3   System.out.println(sdf.format(date));
```

[2] http://mzl.la/1unepot.

This would format a date per the US standard: month/day/year.

More Formatting See SimpleDateFormat[3] for more information.

12.2 Currency

In Java, `Currency` is useful if your code has to deal with currencies in several countries. It provides the following methods:

- `getSymbol()`: Currency symbol for the default locale
- `getSymbol(Locale)`: Currency symbol for the given locale
- `static getAvailableCurrencies()`: Returns the set of available currencies.

For example:

```
1   String pound = Currency.getSymbol(Locale.UK);
```

12.3 TimeZone

In Java 8, time zones are represented by the `java.time.ZoneId` class. There are two types of `ZoneIds`, fixed offsets and geographical regions. This is to compensate for practices such as daylight saving time, which can be very complex.

You can get an instance of a `ZoneId` in many ways, including the following two:

```
1   ZoneId mountainTime = ZoneId.of("America/Denver");
2   ZoneId myZone = ZoneId.systemDefault();
```

To print out all available IDs, use `getAvailableZoneIds()`, as follows:

```
1   System.out.println(ZoneId.getAvailableZoneIds());
```

Write a program that does this and run it.

[3] http://docs.oracle.com/javase/6/docs/api/java/text/SimpleDateFormat.html.

12.4 Scanner

`Scanner` can be used to parse files or user input. It breaks the input into tokens, using a given pattern, which is whitespace by default ("whitespace" refers to spaces, tabs, or anything that is not visible in text).

For example, use the following to read two numbers from the user:

```
1   System.out.println("Please type two numbers");
2   Scanner sc = new Scanner(System.in);
3   int num1 = sc.nextInt();
4   int num2 = sc.nextInt();
```

Write a program that does this and try it out.