Contents

0 Java Object Types

We have seen that Java has eight primitive types: the four integer types, the floating–point types `double` and `float`, the `boolean` type and the `char` type.

Python has a string type; you might ask why we have not given much emphasis to string in Java yet. This is because the string type in Java is *not* a primitive type. It is an example of a Java *object* or *class* type. This distinction is extremely important, because there are significant differences in the behaviors of the two types. We will make a close study of the Java string class and compare its behavior to the string type in Python.

You will see that Java strings have many capabilities. You can slice them as you can Python strings, they know their lengths, and you have access to all characters. You will learn how to use the Java API guide to learn more about any class's capabilities, including those of `String`.

1 Java Strings as a Model for Java Objects

Java handles strings in a manner similar to that of Python. Strings in Java are immutable. Java has an enormous *standard library* containing thousands of classes. The string type is a part of this vast library, an it is implemented in a class called `String`.

Because strings are so endemic in computing, the creators of Java gave Java's string type some features atypical of Java classes, which we shall point out as we progress.

Let us begin by working interactively. Here we see how to read individual characters in a string by their indices.

```
jshell> String x = "abcdefghijklmnopqrstuvwxyz"
x ==> "abcdefghijklmnopqrstuvwxyz"

jshell> x.charAt(0)
$2 ==> 'a'

jshell> x[0]
  Error:
  array required, but java.lang.String found
  x[0]
  ^--^

jshell> x.charAt(25)
$3 ==> 'z'

jshell> x.length()
$4 ==> 26
```

Now let us deconstruct all of this. Strings in Java enjoy an exalted position. The line

```
String x = "abcdefghijklmnopqrstuvwxyz"
```

makes a *tacit* call to `new` and it creates a new `String` object in memory. Only a few other Java class enjoys the privilege of making tacit calls to `new`; these are the *wrapper classes*. Let us take a brief detour to see one of them, `Integer`, which is the wrapper type for the primitive type `int`. You can create an `Integer` either by saying

```
Integer n = 5;
```

or by saying

```
Integer n = new Integer(5);
```

This tacit call to `new` is enabled by a feature called *autoboxing*. We will meet the wrapper classes in full after we finish discussing the string class.

Coming back to our main thread, you can create a string using `new` as well.

```
String x = new String("abcdefghijklmnopqrstuvwxyz");
```

Here we made an *explicit* call to `new`. This is not done very often in practice, as it is excessively verbose, and it can create duplicate copies of immutable objects.

Access to the individual characters within a string is granted via the `charAt` string method. The expression

```
x.charAt(25)
```

can be read as "`x`'s character at index `25`." Just as in Python, the dot (.) indicates the genitive case. The nastygram you got before,

```
|  Error:
|  array required, but java.lang.String found
|  x[0]
|  ^--
```

arises because the square–bracket operator, which exists in Python, is only used to extract array entries in Java. Arrays are a Java data structure, which we will learn about in the next chapter. Finally, we see that a string knows its length; to get it we invoke the `length()` method.

`String` has another atypical feature not found in other Java classes. The operators `+` and `+=` are implemented for Strings. The `+` operator concatenates two strings, just as it does in Python. The `+=` operator works for strings just as it does in Python.

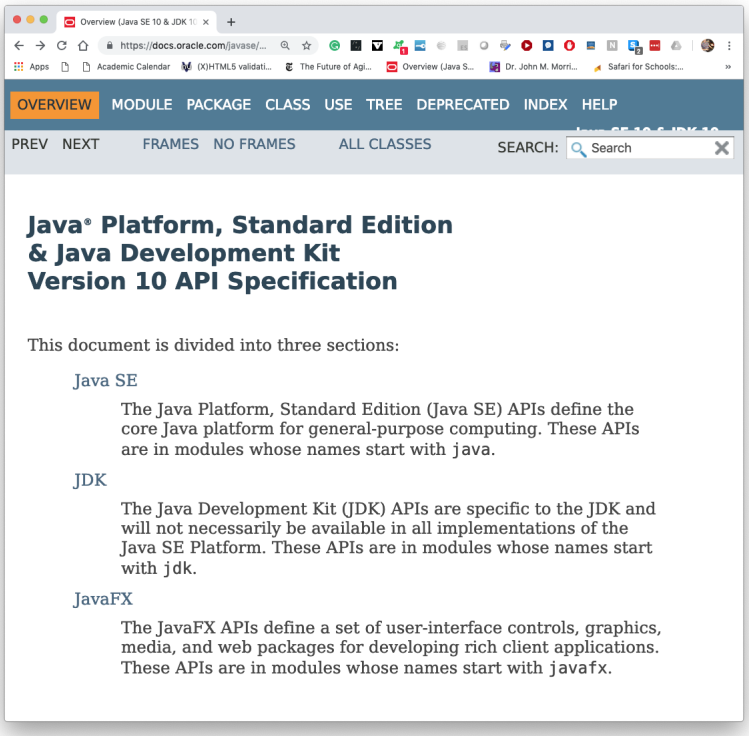
```
jshell> String x = "abc"
x ==> "abc"

jshell> x += "def"
$2 ==> "abcdef"

jshell> x
x ==> "abcdef"
```

Note, however, that the string `"abc"` is not changed. It is orphaned and the String variable `x` now points at `"abcdef"`.

The mechanism of orphaning objects in Java works much as it does in Python. Both Python and Java are garbage-collected languages.

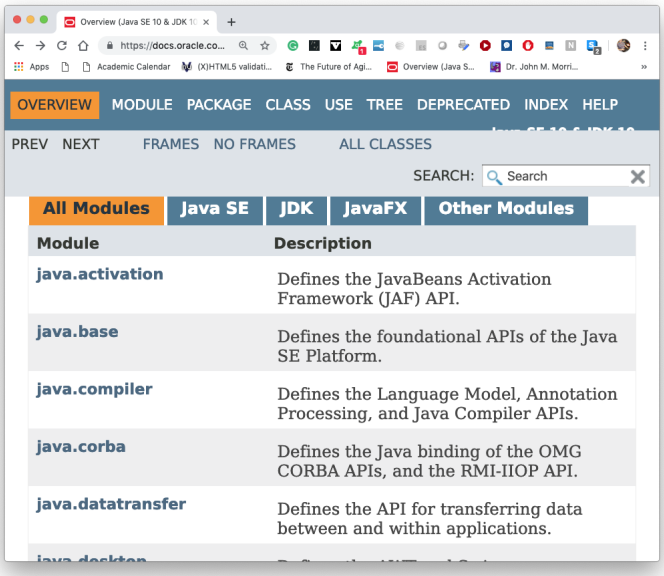
1.1 But is there More?

You might be asking now, "Can I learn more about the Java String class?" Fortunately, the answer is "yes;" it is to found in the Java API (Applications Programming Interface) guide. This is a freely available online comprehensive encyclopedia of all of the classes in the Java Standard Library. Go to this site, `https://docs.oracle.com/javase/10/docs/api/overview-summary.html` and you will see this. There is a link on this page so you can download the entire documentation set onto your computer. When you go to this link, here is what you see.



Classes in Java have two levels of organization. One is modules. We are going to explore the module `java.base`.

Scroll down a little more and you will see the Modules area; it is tabbed. Click on the link for `java.base`.



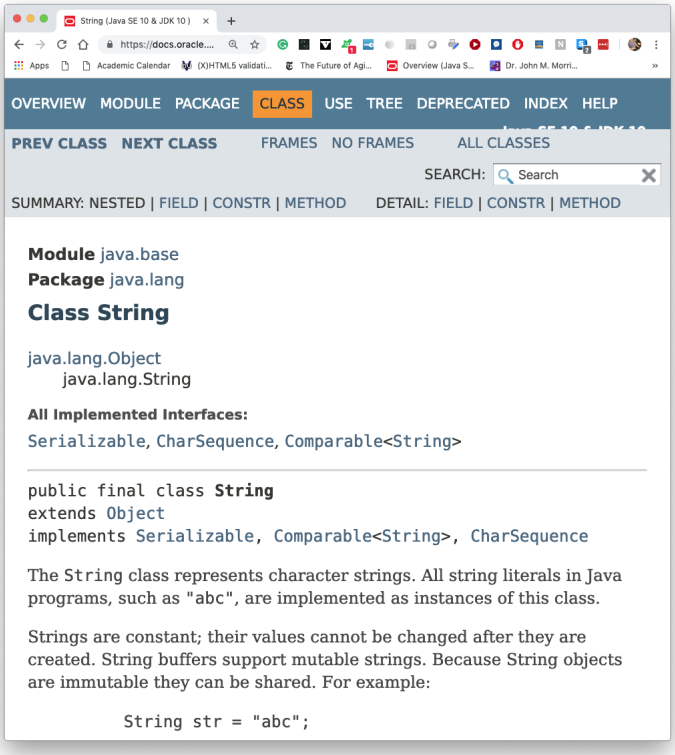
When you open this link, you will see the packages inside of it. Here are some we will use in this book.

`java.lang`	This is the core of the Java language.
`java.io`	This is where Java's fileIO facilities live.
`java.math`	This is the home of `BigInteger`, a class for arbitrary-precision integer arithmetic.
`java.util`	This is where Java's data structures live.

Now click on `java.lang` This page has several segments. Here they are.

- Interfaces
- Classses
- Enums
- Exceptions
- Errors
- Annotation Types

Scroll down the the Classes segment. In it you will find a link for the class `String`. Click on it. This is the top of the page.



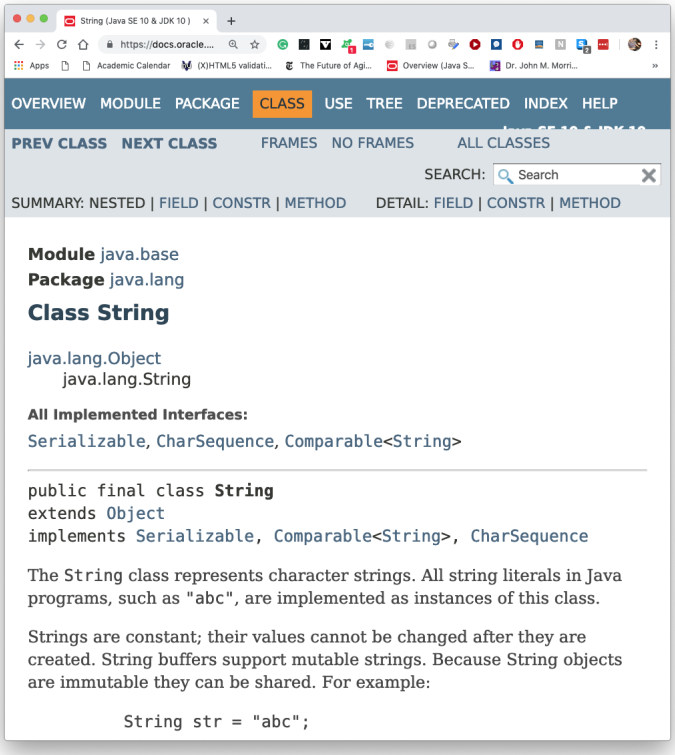
At the very top, the module and the package are identified for you. You can also see the *fully qualified name* of the `String` class, `java.lang.String`. Further down, you see this.

> The String class represents character strings. All string literals in Java programs, such as "abc", are implemented as instances of this class.
>
> Strings are constant; their values cannot be changed after they are created.

In the beginning you will see much that you will not understand. For example, there is this.

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

We will learn about that stuff later. For now, you will learn how to pick out what you need to know. Now scroll down to the Method Summary area. The top of it looks like this.



There are three columns in this table. The first column contains the return type of the method and any modifier (later...). The second shows the name of the method and its argument list. The third describes the method briefly.

To learn more, click on `charAt`. You will see the *method detail*, which looks like this.

```
public char charAt(int index)
```

Returns the char value at the specified index. An index ranges from `0` to `length() - 1`. The first char value of the sequence is at index `0`, the next at index `1`, and so on, as for array indexing. If the char value specified by the index is a surrogate, the surrogate value is returned.

Specified by:
`charAt` in interface `CharSequence`

Parameters:
`index` - the index of the char value.

Returns:
the char value at the specified index of this string. The first char value is at index `0`.

Throws:
`IndexOutOfBoundsException` - if the index argument is negative or not less than the length of this string.

Now click on the link; you will go to the *method detail* for `charAt`. Right after the heading it says

```
public char charAt(int index)
```

This is the method header that appears in the actual `String` class. It then goes on to say the following.

> Returns the `char` value at the specified index. An index ranges from `0` to `length() - 1`. The first char value of the sequence is at index `0`, the next at index `1`, and so on, as for array indexing.

The following paragraph

> If the `char` value specified by the index is a surrogate, the surrogate value is returned.

looks pretty mysterious, so we will ignore it for now. The **Parameters:** heading describes the argument list. **Returns:** heading describes the return value. There are no surprises here.

What is interesting is the **Throws:** heading. This describes run time errors that can be caused by misuse of this method. These errors are not found by the compiler. If you trigger one, your program dies gracelessly and you get great list of nastiness put to your screen. You have observed similar tantrums thrown by Python when you give it an index that is out of bounds in a string, list or tuple.

We shall use this web page in the next sections so keep it open. First it will be necessary to understand a fundamental difference between Java object types and Java primitive types.

Programming Exercises Write a class called `Exercises11` and place the following methods in it.

1. Write the method

```
public boolean isASubstringOf(String quarry, String field)
{
}
```

It should return true when `quarry` is a contiguous substring of `field`. (Think Python `in` construct.)

2. Suppose you have declared the string `cat` as follows.

   ```
   String cat = "abcdefghijklmonopqrstuvwxyz";
   ```

   Find at least two ways in the API guide to obtain the string `"xyz"`. You may use no string literals (stuff inside of `"..."`), just methods applied to the object `cat`. There are at least three ways. Can you find them all?

3. The Python repetition operator `*`, which takes as operands a string and an integer, and which repeats the string the integer number of times does not work in Java. Write a method

   ```
   String repeat(String s, int n)
   ```

   that replicates the action of the Python repetition operator. And yes, there is recursion in Java.

2 Primitive vs. Object: A Case of `equals` Rights

We will study the equality of string objects. A big surprise lies ahead so pay close attention. Create this interactive session in Python. All is reassuringly familiar.

```
>>> good = "yankees"
>>> evil = "redsox"
>>> copy = "yankees"
>>> good == copy
True
>>> good == evil
False
```

No surprises greet us here. Now let us try the same stuff in Java.

```
jshell> String good = "yankees"
good ==> "yankees"

jshell> String evil = "redsox"
evil ==> "redsox"

String copy = new String("yankees")
copy ==> "yankees"
```

```
jshell> good == evil
$4 ==> false

jshell> good == good
$5 ==> true

jshell> good == copy
$7 ==> false
```

Beelzebub! Some evil conspiracy appears to be afoot! Despite the fact that both `good` and `copy` point to a common value of `"yankees"`, the equality test returns a `false`. Now we need to take a look under the hood and see what is happening.

First of all, let's repeat this experiment using integers.

```
jshell> int Good = 5;
Good ==> 5

jshell> int Evil = 4;
Evil ==> 4

jshell> int Copy = 5;
Copy ==> 5

jshell> Good == Evil
$12 ==> false

jshell> Good == Good
$13 ==> true

jshell> Good == Copy
$14 ==> true
```

This seems to be at odds with our result with strings. This phenomenon occurs because primitive and class types work differently.

Without exception, when you use `==` on two variables, you are asking if they store the same value. The value stored by a variable of object type is the memory address of its object. The value stored by a primitive type is the actual value it is storing; *a primitive type variable points directly at its datum*.

So if you are using `==` on variables of object type, you are comparing memory addresses. In our case, Strings do not directly store their object; they store its memory address. this is true of all object types in Java. What a java object knows is a how to find where where the string is stored in memory.

In Python, objects *never* point directly at their datum. Python types such

as `bool`, `float` and `int` are actually immutable objects. This phenomenon is a major difference between Python and Java. Python has no primitive types.

We saw `good == good` evaluate to true because `good` points to the same actual object in memory as itself. We saw `good == copy` evaluate to false, because `good` and `copy` point to separate copies of the string `"yankees"` stored in memory. Therefore the test for equality evaluates to false.

Recall we said that objects have behavior and identity. For objects, the `==` operator is a test for equality of identity. It checks if two objects are in fact one and the same. This behavior is identical to that of the Python `is` keyword, which checks for equality of identity.

What do we do about the equality of strings? Fortunately, the `equals` method comes to the rescue.

```
jshell> good.equals(good)
$15 ==> true

jshell> good.equals(evil)
$16 ==> false

jshell> good.equals(copy)
$17 ==> true
```

Ah, paradise restored... Just remember to use the `equals` method to check for equality of Java objects. This method for strings has a close friend `equalsIgnoreCase` that will do a case-insensitive check for equality of two strings. These comments also apply to the inequality operator `!=`. This operator checks for inequality of identity. To check and see if two strings have unequal values use ! in conjunction with `equals`. Here is an example

```
jshell > !(good.equals(copy))
$ 18 ==> false
>
jshell> !(good.equals(evil))
$ 19 ==> true
```

Finally, notice that Python compares strings lexicographically according to Unicode value by using inequality comparison operators. These do not work in Java. It makes no sense to compare memory addresses. However, the string class has the method

```
int compareTo(String anotherString)
```

We show it at work here.

```
jshell> String little = "aardvark";
little ==> "aardvark"

jshell> String big = "zebra";
big ==> "zebra"

jshell> little <= big
|  Error:
|  bad operand types for binary operator '<='
|    first type:  java.lang.String
|    second type: java.lang.String
|  little <= big
|  ^-----------^

jshell> little.compareTo(big) < 0
$3 ==> true

jshell> little.compareTo(big) == 0
$4 ==> false

jshell> little.compareTo(big) > 0
$5 ==> false
```

You may be surprised `compareTo` returns an integer. However, alphabetical string examples can be done as in the example presented here. This method's sibling method, `compareToIgnoreCase` that does case–insensitive comparisons and works pretty much the same way.

2.1 Aliasing

Consider the following interactive session.

```
jshell> String smith = "granny";
smith ==> "granny"

jshell> String jones = smith;
jones ==> "granny"

jshell> smith == jones
$3 ==> true
```

Here we performed an assignment, `jones = smith`. What happens in an assignment is that the right–hand side is evaluated and then stored in the left. Remember, the string `smith` points at a memory address describing the location where the string `"granny"` is actually stored in memory. So, this memory

address is given to `jones`; both `jones` and `smith` hold the same memory address and therefore both point at the one copy of `"granny"` that we created in memory.

This situation is called *aliasing*. Since strings are immutable, aliasing can cause no harm. We saw in the Python book that aliasing can create surprises.

First we will need to be introduced a property of objects we have omitted heretofore in our discussion: *state*. The state of a string is given by the character sequence it contains. How these are stored is not now known to us, and we really do not need to know or care. We shall tour the rest of the string class in the Java API guide, then turn to the matter of state. Just know that a String's state is specified by the character sequence it contains.

3 More Java `String` Methods

Python's slicing facilities are implemented in Java using `substring`. Here is an example of `substring` at work.

```
jshell> x = "abcdefghijklmnopqrstuvwxyz"
jshell> x.substring(5)
"fghijklmnopqrstuvwxyz"
jshell> x.substring(3,5)
"de"
jshell> x.substring(0,5)
"abcde"
jshell> x
"abcdefghijklmnopqrstuvwxyz"
```

Notice that the original string is not modified by any of these calls; copies are the advertised items are returned by these calls. The `endsWith` method seems pretty self–explanatory.

```
jshell> x.endsWith("xyz")
true
jshell> x.endsWith("XYZ")
false
```

The `indexOf` method allows you to search a string for a character or a substring. In all cases, it returns a -1 if the string or character you are seeking is not present.

```
jshell> x.indexOf('a')
0
jshell> x.indexOf('z')
```

```
25
jshell> x.indexOf('A')
-1
jshell> x.indexOf("bc")
1
```

You can pass an optional second argument to the `indexOf` method to tell is to start its search at a specified index. For example, since the only instance of the character `'a'` in the string `x` is at index 0, we have t

```
jshell> x.indexOf('a', 1)
-1
```

You are encouraged to further explore the `String` API. It contains many useful methods that make strings a useful andx powerful programming tool. The programming exercises here will give you an opportunity to do this.

Programming Exercises Fill in the methods in this class. Copy it and ocmpile it; it will compile in this state. When you are done, it should print all `true`s. Do not worry about the use of the `static` keyword. It makes everything work and it will be explained later. Use the `String` API page to help you.

```
public class Exercises
{
    public static void main(String[] args)
    {
        System.out.println(between("catamaran", 'a').equals("tamar"));
        System.out.println(between("catamaran", 'c').equals(""));
        System.out.println(between("catamaran", 'q').equals(""));
        System.out.println(laxEquals("    boot", "boot"));
        System.out.println(laxEquals("boot    ", "boot"));
        System.out.println(laxEquals("     boot    ", "boot"));
        System.out.println(laxEquals(" \t\n    boot \n\t   ", "boot"));
        System.out.println(getExtension("wd2.tex").equals("tex"));
        System.out.println(getExtension("hello.py").equals("py"));
        System.out.println(getExtension("Hello.java").equals("java"));
        System.out.println(getExtension("tossMeNow").equals(""));
        System.out.println(getExtension(".").equals(""));
        System.out.println(isUpperCaseOnly("EAT NOW 123"));
        System.out.println(!isUpperCaseOnly("eat NOW 123"));
        System.out.println(isUpperCaseOnly(""));
    }
    /*
     * This returns an empty string if q is not present in
```

```
     * s or if it only appears once.  Otherwise, it returns the
     * substring between the first and last instances of q in s.
     */
    public static String between(String s, char q)
    {
        return "";
    }
    /*
     * This returns true if the only difference between s1 and s2
     * is leading or trailing whitespace.
     */
    public static boolean laxEquals(String s1, String s2)
    {
        return false
    }
    /*
     * this returns an empty string if the fileName is empty
     * or has no extension. Otherwise, it returns the extension
     * without the dot.
     */
    public static String getExtension(String fileName)
    {
        return "";
    }
    /*
     * this returns true if the String contains only uppercase
     * or non-alpha characters.
     */
    public static boolean isUpperCaseOnly(String s)
    {
        return false;
    }
}
```

and put these methods in it.

1. Write the method `public Stirng bewteen(String s, char q)` that returns an empty string if `q` occurs once or not at all inside of `s`; otherwise it reteurns the substring in between the first and last instances of `q` in `s`. Examples

4 The Wrapper Classes

Every primitive type in Java has a corresponding *wrapper class*. Such a class "wraps" the primitive object. These classes also supply various useful methods associated with each primitive type. Here is a table showing the wrapper classes.

	Wrapper Classes
Primitive	Wrapper
byte	Byte
short	Short
int	Integer
long	Long
boolean	Boolean
float	Float
double	Double
char	Character

All of these classes have certain common features. You should explore the API guide for each wrapper. They have many helpful features that will save you from reinventing a host of wheels.

One thing you will notice in the wrapper classes is the presence of methods marked `static`. We will discuss this later in more detail, but for now, just know that `static` methods can be called directly using the class name. If you have a class `Foo` and a static method named `cling`, you call it by using `Foo.cling()`. What you don't need to do is this.

```
Foo f = new Foo();
f.cling();
```

You can operate in this way, but calling a static method via the class name is the *preferred* way of calling a static method, and it saves the Java Virtual Machine work.

- **Immutability** Instances of these classes are immutable. You may not change the datum. You may only orphan the object by pointing at a new one with a different datum. This should remind you of Python, because this is how Python treats these types types such at `int`, `bool`, and `float`.
- **A `toString()` method**, which returns a string representation of the datum.
- **A static `toString(primitiveType p)` Method** This method will convert the object passed it into a string. For example, `Integer.toString(45)` returns the string `"45"`.

- **A static parsePrimitive(String s) Method** This method converts a `String` into a value of the primitive type. For example,

  ```
  Integer.parseInt("455")}
  ```

  converts the string `"455"` into the integer `455`. For numerical types, a `NumberFormatException` is thrown if an malformed numerical string is passed them. The `Character` wrapper does not have this feature. You should take note of how this method works in a `Boolean`.
- **Membership in `java.lang`** All of these classes belong to the package `java.lang`; you do not need to import anything to use these classes.

Programming Exercises

1. Write an expression to see if a character is an upper-case letter.
2. Write a method that converts an integer into a comma format string as follows.
   - `456` $\mapsto$ `456`
   - `32768` $\mapsto$ `32,768`
   - `1048576` $\mapsto$ 1,048,576

4.1 Autoboxing and Autounboxing

These features make using the wrapper classes simple. Autoboxing automatically promotes a primitive to a wrapper class type where appropriate. Here is an example. The command

```
Integer i = 5;
```

really amounts to

```
Integer i = new Integer(5);
```

This call to `new` "boxes" the primitive 5 into an object of type `Integer`. The command `Integer i = 5;` automatically boxes the primitive 5 inside an object of type `Integer`. This results in a pleasing syntactical economy.

Autounboxing allows the following sensible–looking code.

```
Integer i = 5;
int x = i;
```

Here, the datum is automatically "unboxed" from the wrapper `Integer` type and it is handed to the primitive type variable `x`.

This is the smart way to compare a primitive with a boxed primitive. Direct comparison can be dangerous and result in errors.

```
jshell> Integer i = 5;
jshell> int y = i;
jshell> int x = 5;
jshell> i == y
true
jshell>
```

Autoboxing and autounboxing eliminate a lot of verbosity from Java; we no longer need to make most `valueOf` and `equals` calls.

5 Two Caveats

Do not box primitive types gratuitously. If you can keep variables primitive without a sacrifice of clarity or functionality, do so. Here is an example of a big mistake caused by seemingly innocuous choice. Although we have not met loops yet, you can easily figure out what is happening here. You are doing a million boxings and unboxings.

```
for(Integer i = 0; i < 1000000; i++)
{
    //code
}
```

This will be a significant performance hit. This is much better.

```
for(int i = 0; i < 1000000; i++)
{
//code
}
```

It is best to prefer the use of primitive types, and to use the wrapper types when you need their helpful methods. We mention them because you will need to use them in conjunction with collections.

Using == on autoboxed primitives is almost always wrong You must use the `equals` method in this case.

The wrappers types become very important when we begin to use collections; you cannot make a collection of primitive objects in Java.

6 Java Classes Know Things: State

So far, we have seen that objects have identity and that they have behavior, which is reflected by a class's methods.

We then saw that a string "knows" the character sequence it contains. We do not know how that sequence is stored, and we do not need to know that. The character sequence held by a string is reflective of its *state*. The state of an object is what an object "knows." Observe that the outcome of a Java method on an object can, and often does, depend upon its state.

To give you a look behind the scenes, we shall now produce a simple class which will provides a blueprint for making objects having state, identity and behavior. To do this, it will necessary to introduce some new ideas in Java, the *constructor* and *method overloading*.

Place the following code in a file, compile and save it with the name `Point.java`. We are going to create a simple class for representing points in the plane with integer coördinates.

```
public class Point
{
}
```

What does such a point need to know? It needs to know its `x` and `y` coördinates. Here is how to make it aware of these.

```
public class Point
{
    private int x;
    private int y;
}
```

You will see a new keyword: `private`. This says that the variables `x` and `y` are not visible outside of the class. These variables are called *instance variables* or *state variables*. We shall see that they specify the state of a `Point`.

Why this excessive modesty? Have you ever bought some electronic trinket, turned it upside–down and seen "No user serviceable parts inside" emblazoned on the bottom? The product–liability lawyers of the trinket's manufacturer figure that an ignorant user might bring harm to himself whilst fiddling with the entrails of his device. Said fiddling could result in a monster lawsuit that leaves the principles of the manufacturer living in penury.

Likewise, we want to protect the integrity of our class; we will not allow the user of our class to monkey with the internal elements of our program. We will

permit the client programmer access to these elements by creating methods that give access. This is a hard-and-fast rule of Java programming: *Always declare your state variables private.*

Additionally, if we decide later that it is better to implement the class in a newer and better way, we can do this and we can keep the interface the same. This allows us to do an "engine upgrade" but have the operation of the car be the same. It will just have a little more pep when you step on the gas.

Now compile your class. Let us make an instance of this class and deliberately get in trouble.

```
jshell> Point p = new Point();
p ==> Point@26653222

jshell> p.x
|  Error:
|  x has private access in Point
|  p.x
|  ^-^
```

We have debarred ourselves from having any access to the state variables of an instance of the `Point` class. This makes our class pretty useless. How do we get out of this pickle?

6.1 Quick! Call the OBGYN! And get a load of `this`!

Clearly a `Point` needs help initializing its coördinates. For this purpose we use a special method called a *constructor*. A constructor has no return type; in fact it is the only method in a class which can lack a return type. When the constructor is finished, good programming practice dictates that all state variables should be initialized. Constructors are OBGYNs: they oversee the birth of objects.

Now for some grammatical ground rules. The constructor for a class must have the same name as the class. In fact, only constructors in a class may have the same name as the class. We now write a constructor for our `Point` class.

```
public class Point
{
    private int x;
    private int y;
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
```

```
    }
}
```

When you are programming in a class, you *are* that object. The keyword `this` refers to "me." The dot construct is the genetive case, so `this.x` is "my `x`."

Now compile your class. To make a point at (3,4), call the constructor by using `new`. The `new` keyword calls the class's constructor and oversees the birth of an object.

```
jshell> Point p = new Point(3,4);
p ==> Point@26653222

jshell> Point q = new Point();
|  Error:
|  constructor Point in class Point cannot be applied to given types;
|    required: int,int
|    found: no arguments
|    reason: actual and formal argument lists differ in length
|  Point q = new Point();
|            ^---------^
```

The `Point p` is storing the point (3,4). Remember, the variable `p` itself only stored a memory address. The point (3,4) is stored at that address.

One other thing we see is that once we create a constructor the *default constructor*, which has an empty signature, no longer exists.

Note the similarity of this process to the Python Point class we created earlier. The Python `__init__` method behaves much like a Java constructor; it is called every time a new Python object of type `Point` is created.

```
import math
class Point(object):
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y

p = Point()
print ("p = ({0}, {1})".format(p.x, p.y))
q = Point(3,4)
print ("q = ({0}, {1})".format(q.x, q.y))
```

Now go back to the `String` class in the API guide. Scroll down to the constructor summary; this has a blue header on it and it is just above the method summary. You will see that the string class has many constructors.

How is this possible? We faked in in Python by using default arguments. Can we do this for our point class in Java?

Happily, the answer is "yes".

6.2 Method and Constructor Overloading

The *signature* of a method is an ordered list of the types of its arguments. Java supports *method overloading*: you may have several methods bearing the same name, provided they have different signatures. This is why you see several versions of `indexOf` in the `String` class. Java resolves the ambiguity caused by overloading at compile time by looking at the types of arguments given in the signature. It looks for the method with the right signature and it then calls it.

Notice that the static typing of Java allows it to support method overloading. Python fakes method overlaoding via the facility of default arguments. Here is another example of Python default arguments at work.

```
def f(x = 0, y = 0, z = 0):
        return x + y + z
print "f() = ", f()
print "f(3) = ", f(3)
print "f(3, 4) = ", f(3, 4)
print "f(3, 4, 5) = ", f(3, 4, 5)
```

```
unix> python overload.py
f() =  0
f(3) =  3
f(3, 4) =  7
f(3, 4, 5) =  12
unix>
```

You can use this principle on constructors, too. Let us now go back to our `Point` class. We will make the default constructor (sensibly enough) initialize our point to the origin.

```
public class Point
{
    private int x;
    private int y;
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
```

```
    public Point()
    {
        this.x = 0;
        this.y = 0;
    }
}
```

Compile this class. Then type in this interactive session.

```
jshell> Point p = new Point(3,4);
p ==> Point@26653222

jshell> Point q = new Point();
q ==> Point@68c4039c
```

Voila! The default constructor is now working.

6.3 Get a load of `this` again!

The eleventh commandment reads, "Thou shalt not maintain duplicate code." This sounds Draconian, but it is for reasons of convenience and sanity. If you want to modify your program, you want to do the modifications in ONE place. Having duplicate code forces you to ferret out every copy of a duplicated piece of code you wish to modify. You should strive to avoid this.

One way to avoid it is to write separate methods to perform tasks you do frequently. Here, however, we are looking at our constructor. You see duplicate code in the constructors. To eliminate it, you may use the `this` keyword to call one constructor from another. We shall apply `this` here.

```
public class Point
{
    private int x;
    private int y;
    public Point(int _x, int _y)
    {
        this.x = x;
        this.y = y;
    }
    public Point()
    {
        this(0,0);
    }
}
```

6.4 Now Let Us Make this Class DO Something

So far, our `Point` class is devoid of features. We can create points, but we cannot see what their coördinates are. Now we shall provide *accessor methods* that give access to the cöordinates. While we are in here we will also write a special method called `toString`, which will allow our points to print nicely to the screen.

First we create the accessor methods. Here is how they should work.

```
jshell> Point p = new Point(3,4);
p ==> Point@26653222

jshell> Point q = new Point();
q ==> Point@68c4039c

jshell> p.getX()
$4 ==> 3

jshell> p.getY()
$5 ==> 4

jshell> q.getX()
$6 ==> 0

jshell> q.getY()
$7 ==> 0
```

Making them is easy. Just add these methods to your `Point` class.

```
public int getX()
{
    return x;
}
public int getY()
{
    return y;
}
```

These accessor or "getter" methods allow the user of your class to see the coördinates of your `Point` but the user cannot use the getter methods to change the state of the `Point`. So far, our point class is immutable. There is no way to change its state variables, only a way to read their values.

To get your points to print nicely, create a `toString` method. Its header *must* be

```
public String toString()
```

In this method we will return a string representation for a point. Place this method inside of your `Point` class.

```
    public String toString()
    {
        return "("+ x + ", " + y + ")";
    }
```

Compile and run. The `toString()` method of an object's class is called *automatically* whenever you print an object. Every Java object is born with a `toString()` method. We saw that this built–in method for our point class was basically useless. By implementing the `toString` method in our class, we are customizing it for our purposes. Here we see our nice representation of a `Point`.

```
jshell> Point p = new Point(3,4)
(3, 4)
jshell> System.out.println(p)
(3, 4)
jshell>
```

You will see that many classes in the standard library customize this method.

Now let us write a method that allows us to compute the distance between two points. To do this we will need to calculate a square-root. Fortunately, Java has a scientific calculator. Go to the API guide and look up the class `Math`. To use a `Math` function, just prepend its name with `Math.`; for example `Math.sqrt(5)` computes $\sqrt{5}$. All of the methods of this class are static. Many of the names of these functions are the same as they are in Python's `math` library and in and C/C++'s `cmath` and `math.h` libraries.

Now add this method to your class. You will see that it is just carrying out the distance formula between your point `(x,y)` and the point `q`.

```
    public double distanceTo(Point q)
    {
        return Math.sqrt( (x - q.x)*(x - q.x) + (y - q.y)*(y - q.y));
    }
```

Compile and run.

```
jshell> Point origin = new Point()
(0, 0)
jshell> Point p = new Point(5,12);
jshell> origin.distanceTo(p)
```

```
13.0
jshell> p.distanceTo(origin)
13.0
jshell>
```

Programming Exercises Here is a chance to try out some new territory in Python. Python has special methods called *hooks* that do special jobs. We have met the `init` hook; all hooks are surrounded by double-underscores.

1. The Python hook `__str__` tells Python to represent a Python object as a string. Its method header is

   ```
   def __str__(self):
   ```

   Make a method for the Python Point class that represents a Point as a string.

2. The Python hook `__eq__` can check for equality of objects. You can cause to points to be compared using `==` with this hook. Its header is

   ```
   def __eq__(self, other):
   ```

   Implement this for our Python `Point` class. What header do you decide to use?

3. Look up the `hypot()` method in the `Math` class. How can you use it in our `Point` class?

6.5 Who am I?

In Java there are two programming roles: that of the class developer and that of the client programmer. You assume both roles, as all code in Java lives inside of classes. You are the class developer of the class you are writing, and the client programmer of the classes you are using. Any nontrivial Java program involves at least two classes, the class itself and often the class `String` or `System.out`. In practice, as you produce Java classes, you will often use several different classes that you have produced or from the Java Standard Library.

So while we are creating the `Point` class, we should think of ourselves as *being* `Point`s. A `Point`s knows its coördinates. Since you are a point when programming in the class `Point`, you have access to your private variables. You also have access to the private variables of any instance of class `Point`. This is why in the `distanceTo` method, we could use `q.x` and `q.y`.

In the last interactive session we made two points with the calls

```
Point origin = new Point();
Point p = new Point(5,12)
```

This resulted in the creation of two points. The call

```
p.distanceTo(origin)
```

returned 13.0. What is says is "Call `p`'s `distanceTo` method using the argument `origin`." In this case, you should think of "you" as `p`. The point `origin` is playing the role of the point `q` in the method header. Likewise, the call

```
origin.distanceTo(p)
```

is calling `p`'s distance to `origin`. In the first case, "I" is `origin`, in the second, "I" is `p`.

6.6 Mutator Methods

So far, all of our class methods have only looked at, but have not changed, the state of a point object. Now we will make our points mutable. To this end, create two "setter" methods and place them in your `Point` class.

```
    public void setX(int a)
    {
        x = a;
    }
    public void setY(int b)
    {
        y = b;
    }
```

Now compile and type in the following.

```
jshell> p = new Point()
(0, 0)
jshell> p.setX(5)
jshell> p
(5, 0)
jshell> p.setY(12)
jshell> p
(5, 12)
```

Our point class is now mutable: We are now giving client programmers permission to reset each of the coördinates. These new methods are called "mutator" methods, because they change the state of a `Point` object. Instances of our `Point` class are mutable, much as are Python lists. Mutability can be convenient, but it can be dangerous, too. Watch us get an ugly little surprise from aliasing.

To this end, let us continue the interactive session we started above.

```
jshell> q = p;
jshell> q
(5, 12)
jshell> q.setX(0)
jshell> p
(0, 12)
jshell> q
(0, 12)
```

Both `p` and `q` point at the same object in memory, which is initially storing the point `(5,12)`. Now we say, "`q`, set the x–coördinate of the point you are pointing at to 0. Well, `p` happens to be pointing at precisely the same object. In this case `p` and `q` are aliases of one another. If you call a mutator method from either variable, it changes the value pointed at by the other!

If we wanted `p` and `q` to be independent copies of one another, a different procedure is required. Let us now create a method called `clone`, which will return an independent copy of a point.

```
public Point clone()
{
    return new Point(x,y);
}
```

Compile and fire up a new `jshell` session. Now we will test–drive our new `clone` method. We will make a point `p`, an alias for the point `alias`, and a copy of the point `copy`.

```
jshell> Point p = new Point(3,4)
(3, 4)
jshell> Point alias = p
(3, 4)
jshell> Point copy = p.clone();
jshell> p
(3, 4)
```

Continuing, let us check all possible equalities.

```
jshell> p == alias
true
jshell> copy == alias
false
jshell> p == copy
false
```

We can see that `p` and `q` are in fact aliases the same object, but that `alias` is not synonymous with either `p` or `q`.

```
jshell> p
(3, 4)
jshell> alias
(3, 4)
jshell> copy
(3, 4)
```

All three point at a point stored in memory that is `(3,4)`. Now let us call the mutator `setX` on `p`; we shall then inspect all three.

```
jshell> p.setX(0)
jshell> p
(0, 4)
jshell> alias
(0, 4)
jshell> copy
(3, 4)
```

The object pointed to by both `p` and `q` was changed to `(0,4)`. The copy, however, was untouched.

Look at the body of the `clone` method. It says

```
return new Point(x,y);
```

This tells Java to make an entirely new point with coördinates `x` and `y`. The call to `new` causes the constructor to spring into action and stamp out a fresh, new `Point`.

7 The Scope of Java Variables

In this section, we shall describe the lifetime and visibility of Java variables. The rules differ somewhat from Python, and you will need to be aware of these differences to avoid unpleasant surprises.

There are two kinds of variables in Java, state variables and *local* variables. Local variables are variables created inside of any method in Java. State variables are visible anywhere in a class. Where they are declared in a class is immaterial, but you should declare them at the top of your class. This makes them easy to find and manage. You could move them to the end of the class with no effect.

The rest of our discussion pertains to local variables. All local variables in Java have a *block*; this is delimited by the closest pair of matching curly braces containing the variable's declaration. The first rule is that *no local variable is visible outside of its block.* The second rule is that *a local variable is not visible until it is created.* You will notice that these rules are stricter than those of Python. As in Python, variables in Java are not visible prior to their creation; this rule is exactly the same.

Here is an important difference. Variables created inside of Python functions are visible from their creation to the end of the function, even if they are declared inside of a block in that function. Here is a quick example in a file named `laxPyScope.py`.

```
def artificialExample(x):
    k = 0
    while k < len(x):
        lastSeen = x[k]
         k += 1
    return lastSeen
x = "parfait"
print "artificialExample(" + x + ") = ", artificialExample(x)
```

It is easy to see that the function `artificialExample` simply returns the last letter in a nonempty string. We run it here.

```
$ python laxPyScope.py
artificialExample(parfait) =  t
$
```

Observe that the variable `lastSeen` was created inside a block belonging to a `while` loop. In Java's scoping rules, this variable would no longer be visible (it would be destroyed) as soon as the loop's block ends.

There are some immediate implications of this rule. Any variable declared inside of a method in a class can only be seen inside of that method. That works out the same as in Python. Let us code up exactly the same thing in Java in a class `StrictJavaScope`. In this little demonstration, you will see Java's `while` loop at work.

```
public class StrictJavaScope
{
    public char artificialExample(String x)
    {
        int k = 0;
        while( k < x.length())
        {
```

```
            char lastSeen = x.charAt(k);
            k += 1;
        }
        return lastSeen;
    }
}
```

Now compile and brace yourself for compiler grumblings.

```
 javac StrictJavaScope.java
StrictJavaScope.java:11: error: cannot find symbol
        return lastSeen;
               ^
  symbol:   variable lastSeen
  location: class StrictJavaScope
1 error
```

Your symbol `lastSeen` died when the `while` loop ended. Even worse, it got declared each time the loop was entered and died on each completion of the loop.

How do we fix this? We should declare the `lastSeen` variable before the loop. Then its block is the entire function body, and it will still exist when we need it. Here is the class with repairs effected.

```
public class StrictJavaScope
{
    public char artificialExample(String x)
    {
        int k = 0;
        char lastSeen = ' ';
        while( k < x.length())
        {
            lastSeen = x.charAt(k);
            k += 1;
        }
        return lastSeen;
    }
}
```

Peace now reigns in the valley.

```
jshell> s = new StrictJavaScope();
jshell> s.artificialExample("parfait")
't'
jshell>
```

`while` We are at it The use of the `while` loop is entirely natural to us and it looks a lot like Python. There are some differences and similarities. The differences are largely cosmetic and syntactical. The semantics are the same, save of this issue of scope we just discussed.

- **similarity** The `while` statement is a boss statement. No mark occurs in Java at the end of a boss statement.
- **difference** Notice that there is NO colon or semicolon at the end of the `while` statement. Go ahead, place a semicolon at the end of the `while` statement in the example class. It compiles. Run it. Now figure out what you did, Henry VIII.
- **difference** Notice that predicate for the `while` statement is enclosed in parentheses. This is required in Java; in Python it is optional.
- **similarity** The `while` statement owns a block of code. This block can be empty; just put an empty pair of curly braces after the loop header.

The scoping for methods and state variables is similar. State variables have *class scope* and they are visible from anywhere inside of the class. They may be modified by any of the methods of the class. Any method modifying a state variable is a mutator method for the class. Be careful when using mutator methods, as we have discussed some of their perils when we talked about aliasing. A good general rule is that if a class creates small objects, give it no mutator methods. For our `Point` class, we could just create new `Point`s, rather than resetting coördinates. Then you do not have to think about aliasing. In fact, it allows you to share objects among variables freely and it can save space. It also eliminates the need for copying objects.

Later, we will deal with larger objects, like graphics windows and displays. We do not want to be unnecessarily calling constructors for these large objects and we will see that these objects in the standard library have a lot of mutator methods.

All methods are visible inside of the class. To get to methods outside of the class, you create an instance of the class using `new` and call the method via the instance. Even if your state variables are (foolishly) `public`, you must refer to them via an instance of the class. Let us discuss a brief example to make this clear.

Suppose you have a class `Foo` with a method called `doStuff()` and public a public state variable `x`. Then to get at `doStuff` or `x` we must first create a new `Foo` by calling a constructor. In this example we will use the default.

```
Foo f = new Foo();
```

Then you can call `doStuff` by making the call

```
f.doStuff();
```

Here you are calling `doStuff` via the instance `f` of the class `Foo`. To make `f`'s `x` be 5, we enter the code

```
f.x = 5;
```

Notice that the "naked" method name and the naked variable name are not visible outside of the class. In practice, since all of our state variables will be marked `private`, no evidence of state variables is generally visible outside of any class.

8 The Object-Oriented Weltanschauung

Much emphasis has been placed here on classes and objects. In this section we will have a discussion of programming using objects. We will begin by discussing the procedural programming methods we developed in Chapters 0-7 of the Python book.

8.1 Procedural Programming

When we first started to program in Python, we wrote very simple programs that consisted only of a main routine. These programs carried out small tasks and were short so there was little risk of confusion or of getting lost in the code.

As we got more sophisticated, we began using functions as a means to break down, or modularize, our program into manageable pieces. We would then code each part and integrate the functions into a program that became a coherent whole. Good design of programs is "top down." You should nail down what you are trying to accomplish with your program. Then you should break the program down into components. Each component could then be broken into smaller components. When the components are of a manageable size, you then code them up as functions.

To make this concrete, let us examine the case of writing a program that accepts a string and which looks through an English word list, and which shows all anagrams of the string you gave as input which appear in the word list.

To do this, you could write one monolithic procedure. However, the procedure would get pretty long and it would be trying to accomplish many things at once. Instead we might look at this and see we want to do the following

- Obtain the word from the user.
- Lower-case the word and permute the letters so they are in alphabetical order

- Open a word list file.
- for each word in the list:
  - Lower-case each word in the wordlist and put it in alphabetical order.
  - If you get a match, put the word on an output list
- Return the list of words we obtained to the user.

Not all the tasks here are of the same difficulty. The first one, obtain the word from the user, is quite easy to do. We, however have to make a design decision and decide how to get the word from the user. This is a matter of deciding the program's *user interface*.

Python is an object-oriented language like Java with a library of classes. Many of the Python classes can save you great gobs of work; the same is true in Java. Always look for a solution to your problem in the standard library before trying to solve it yourself! If you create classes intelligently, you will see that you will be able to reuse a lot of code you create.

Returning to our problem, you would revisit each sub-problem you have found. If the sub-problem is simple enough to write a function for it, code it. Otherwise, break it down further.

This is an example of top–down design for a *procedural* program. We keep simplifying procedures until they become tractable enough to code in a single function. We program with *verbs*.

The creation of functions gives us a layer of abstraction. Once we test our functions, we use them and we do not concern ourselves with their internal details (unless a function goes buggy on us), we use them for their *behavior*. Once a function is created and its behavior is known, we no longer concern ourselves with its local variables and the details of its implementation.

This is an example of *encapsulation*; we are thinking of a function in terms of its behavior and not in terms of its inner workings.

8.2 Object–Oriented Programming

In object–oriented programming, we program with *nouns*. A class is a sophisticated creature. It creates *objects*, which are computational creatures that have state, identity and behavior. We shall see here that encapsulation plays a large role in object-oriented programming. Good encapsulation dictates that we hide things from client programmers they do not need to see. This is one reason we make our state variables private. We may even choose to make certain methods

private, if the do think they are of real use other than that of a service role the other methods of the class.

You still do top-down design, but you begin by thinking about what kind of objects the task at hand entails. This prompts you to think about the classes you can use from the standard libraries and those you need to write yourself. You must think about the ways in which they interact.

For each class, you have to think about what state it needs to maintain, and what methods it should have so it can do its job properly. We arrive here in much more complex world than that of procedural programming.

When you used the `String` class, you did not need to know how the characters of a `String` are stored. You do not need to know how the `substring()` method works: you merely know the behavior it embodies and you use it. What you can see in the API guide is the *interface* of a class; this is a class's public portion. You are a client programmer for the entire standard library.

What you do not see is the class's *implementation*. You do not know how the `String` class works internally. You could make a good guess. It looks as if a list of characters that make up the string is stored somewhere. That probably reflects the state of a `String` object.

A string, however, is a fairly simple object. The contents of a window in a graphical user interface (GUI) in Java is stored in an object that is an instance of the class `Stage`. How do you store such a thing? Is it different on different platforms? All of a sudden we feel the icy breath of the possibly unknown.... However, there is nothing to fear! In Java the `Stage` class has behaviors that allow you to work with a frame in a GUI and you do not have to know *how* the internal details of the `Stage` work. This is the beauty of encapsulation. Those details are thankfully hidden from us!

For an everyday example let us think about driving a car. You stick in the key, turn it, and the ignition fires the engine. You then put the car in gear and drive. Your car has an interface. There is the shifter, steering wheel, gas pedal, the music and climate controls, the brakes and the parking brake. There are other interface items such as door locks, seat adjusters, and the dome light switch.

These constitute the "public" face of your car. You work with this familiar interface when driving. It is similar across cars. Even if your car runs on diesel, the interface you use to drive is not very different from that of a gasoline–fueled car.

You know your car has "private parts" to which you do not have direct access when driving. Your gas pedal acts as a mutator method does; when you depress it, it causes more gas to flow to the fuel injection system and causes the RPM of the engine to increase. The RPM of the engine is a state variable for your car. Your tachometer (if your car has one) is a getter method for the RPM of your engine. You affect the private parts (the implementation) of your car only

indirectly. Your actions occur through the interface.

Let's not encapsulate things for a moment. Imagine if you had to think about *everything* your car does to run. When you stick your key in the ignition, if you drive a newer car, an electronic system analyzes your key fob to see if your key is genuine. That then triggers a switch that allows the ignition to start the car. Then power flows to the starter motor..... As you are tooling down the highway, it is a safe bet you are not thinking about the intricacies of your car's fuel injection system and the reactions occurring in its catalytic converter. You get the idea. Encapsulation in classes simplifies things to a manageable essence and allows us to think about the problem (driving here) at hand. You use the car's interface to control it on the road. This frees your mind to think about your actual driving.

So, a Java program is one or more classes working together. We create instance of these classes and call methods via these instances to get things done. In the balance of this book, you will gain skill using the standard library classes. You will learn how to create new classes and to create extensions of existing ones. This will give you a rich palette from which to create programs.

Exercises

1. Think about your bicycle. What constitutes its interface?
2. What is the interface to your computer? How do you interact with it and control it? What are some of its "private parts"?
3. How about a takeout pizza joint? How do you interact with it? What are some of its public and private parts?