Chapter 1, Introducing Java

John M. Morrison

November 11, 2019

Contents

0 Welcome to JShell!

If you run Python with no file, you get an interactive Python shell. Java has a similar feature named `jshell`. To fire up `jshell`, type `jshell` at the command prompt and you will see this.

```
$ jshell
> jshell
|  Welcome to JShell -- Version 9.0.1
|  For an introduction type: /help intro

jshell>
```

JShell has command that allows you to see the symbol table for your session, obtain help, and quit. We will use it quite a bit for inspecting and exploring classes.

Here is a summary of the commands. All of them begin with a /.

- `/exit` This quits `jshell`.
- `/help` This gives help on commands. The ussage is
- `/imports` This lists all classes you have imported.
- `/list` This lists all snippets you have entered.
- `/open` This reads a class into `jshell` so you can inspect it.
- `/types` This lists all active classes in the session.
- `/vars` This lists all variables and their values `/help /command` and you get a little man page for that command.

Here we show how to quit `jshell` using the `/exit` command.

```
$ jshell
> jshell
|  Welcome to JShell -- Version 9.0.1
|  For an introduction type: /help intro

jshell> /exit
|  Goodbye
$
```

You can also quit it by typing control-d.

You enter pieces of code called *snippets* into the shell and `jshell` runs them. Let's do that and test-drive some of the commands. Begin by entering some variable declarations.

```
jshell> int x = 4;
x ==> 4

jshell> String y = "spaghetti";
y ==> "spaghetti"
```

```
jshell> int z = 15;
z ==> 15
```

Now, watch `jshell` evaluate an expression.

```
jshell> x*z
$4 ==> 60
```

The symbol `$4` is a valid variable in your `jshell` session.

```
jshell> $4
$4 ==> 60
```

Now we try the `/vars` command.

```
jshell> /vars
|    int x = 4
|    String y = "spaghetti"
|    int z = 15
|    int $4
```

Ooh, yummy, here is our visible symbol table. Now let's make a `/list` and check it twice.

```
jshell> /list
   1 : int x = 4;
   2 : String y = "spaghetti";
   3 : int z = 15;
   4 : x*z
   5 : $4
```

Here we see all of the snippets we have created this session. Next let us try `/vars`. Now let us quit this session. We will show how to inspect a class. Create this class.

```
public class Example
{
    public void go()
    {
        int x = 5;
        System.out.println("x = " + x);
    }
}
```

Next crank up `jshell` and open the file and inspect a method as follows.

```
jshell> /open Example.java

jshell> Example e = new Example();
e ==> Example@26653222

jshell> e.go()
x = 5
```

Now see the `/types` command in action.

```
jshell> /types
|    class Example
```

1 How does Java Work on a Mechanical Level?

We will begin by looking at the mechanics of producing a program. We will then sketch a crude version of what actually happens during the process and refine it as we go along. Here is a simplified life-cycle for a Java program. So you can follow along, make this empty class. `Foo.java`.

```
public class Foo
{
}
```

1. **Edit** You begin the cycle by creating code in a text editor and saving it. Each file of Java will have a `public` class in it. The class is the fundamental unit of Java code; all of your Java programs will be organized into classes. Java classes are similar to those in Python; later we will compare them. The name of the class must match the name of the file; otherwise you will get a nastygram from the compiler. As you saw in the example at the end of the last section, the file containing `public class Foo` must be be named `Foo.java`; failure to adhere to this convention will be punished by the compiler.
   Deliberately trigger this error creating an empty class `Right` in the file `Wrong.java`. Compile and you will receive this beating.

   ```
   $ javac Wrong.java
   Wrong.java:1: error: class Right is public,
       should be declared in a file named Right.java
   public class Right
          ^
   1 error
   $
   ```

An optional but nearly universal convention is to capitalize class names. You should adhere to this rule in the name of wise consistency. This is done by all serious Java programmers; uncapitalized class names just confuse, annoy, and vex others.

2. **Compile** Java is an example of a *high–level language*. A complex program called a *compiler* converts your program into an executable form.
   Compilation at the UNIX command line in Java is simple. To compile `Foo.java`, proceed as follows

   ```
   $ javac Foo.java
   ```

   When done, list your files and you should see `Foo.class`. If your program contains syntactical errors that make it unintelligible to the compiler, the compilation will abort. When this happens, nothing executes and no executable file is generated. In contrast, in the Python language, the program stops running when an syntactical error is encountered; in Java the program does not run at all unless it compiles successfully. There is no compile time in Python.
   If your program does not compile, you will get one or more error messages. These will be put to `stderr`, which by default, is your terminal window. You will need to re–edit your code to stamp out these errors before it will compile.
   Java compiles programs in the machine language of the *Java Virtual Machine*; this machine is a virtual computer that is built in software. Its machine language is called *Java byte code*. In the `Foo.java` example, successful compilation yields a file `Foo.class`; this file consists of Java byte code. Your JVM takes this byte code and converts it into machine language for your particular platform, and your program will run. Java is not the only language that compiles to the JVM. Others include Scala, Clojure, a Lisp dialect, Processing, an animation language created in the MIT media lab, and Groovy, a scripting language. There is even a JVM implementation of Python called Jython.

3. **Run** For a program to run, it needs a special method called a *main method*. This method goes inside your class and it looks like this.

   ```
   public class Foo
   {
       public static void main(String[] args)
       {
           System.out.println("foo");
       }
   }
   ```

   Save and compile this. If your compilation succeeds, you will be able to run your program as follows.

   ```
   $ javac Foo.java
   $ java Foo
   foo
   ```

You will run your program and see if it works as specified. If not... back to the step **Edit**. You can have errors at run time, too. These errors result in an "exploding heart;" these ghastly things are nasty error messages containing many lines of ugly incantations. You can also have logic errors in your program; in this case, the program will reveal some unexpected behavior you did not want.

You will often hear the terms "compile time" and "run time" used. These are self-explanatory. Certain events happen when your program compiles, these are said to be compile time events. Others happen at run time. These terms will often be used to describe errors.

It is a good idea to compile any class before attempting to inspect it in `jshell`. If it fails to compile, your `jshell` session will be polluted with error messages.

Next, we take a brief tour of Python classes.

2 Python Classes and Objects

Python classes have a very simple structure; we will take a quick look at these before wading into Java classes. You can create a Python class with two lines of code.

```
class Simple(object):
    pass
```

A class is a blueprint for creating objects; this principle works the identically Python and Java. Making an object from this class is ... simple. Just do this.

```
>>> class Simple(object):
...     pass
...
>>> s = Simple()
>>> t = Simple()
>>>
```

We have created two objects `s` and `t` that are *instances* of this class.

We have learned that objects have state, identity and behavior. Recall that state is what an object knows, behavior is what an object does, and identity is what an object is (a hunk of memory). Since the body of our class is empty, `Simple` objects know nothing and do nothing of great use. The can, however do some basic stuff. They can be represented as strings, and they can be checked for equality. Here we see this.

```
>>> s
<__main__.Simple object at 0x12e3250>
>>> t
<__main__Simple object at 0x12e3290>
>>> s == t
False
>>>
```

In Python, you can attach state to objects. Watch what is happening here.

```
>>> s.x = "I am x."
>>> s.y = "I am y."
>>> s.x
'I am x.'
>>> t.x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Simple' object has no attribute 'x'
>>>
```

Now the object `s` knows its `x` and `y`, but `t` still knows nothing. We can make all instances of our class know `x` and `y` as follows.

```
>>> class Simple:
...     x = "I am x."
...     y = "I am y."
...
>>> s = Simple()
>>> t = Simple()
>>> s.x
'I am x.'
>>> t.x
'I am x.'
>>> s.y
'I am y.'
>>> t.y
'I am y.'
>>>
```

So far, our classes have fixed state. Objects of type `Simple` all have the same `x` and `y`. This is not terrible useful. Suppose we want to make a `Point` class to represent points in the plane with integer coördinates. When we create a `Point`, we might want to specify its coördiantes. To do this, we will use a special method called `__init__`, which runs immediately after the object is created.

Let us now consider this program.

```
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 run this program and see the following.

```
$ python3 Point.py
p = (0, 0)
q = (3, 4)
```

We see a lot of new stuff here, so let us go through it with some care. We know that the `__init__` method runs immediately after a Point object is created. Its argument list is `(self, x, y)`. The purpose of the `x` and `y` seem clear: they furnish coördiantes to our `Point` object.

We also see this `self`. What is this? When you program in the `Point` class, you *are* a `Point`. So `self` is you. In the statement `self.x = x`, you are attaching the value `x` sent by the caller to yourself. The quantities `self.x` and `self.y` constitute the state of an instance of this class. This is how a `Point` knows its coördinates. The symbols `self.x` and `self.y` have scope inside of the entire class body. Take note that all argument lists of methods in a Python class must begin with `self`.

Now let us make a `Point` capable of doing something. Modify your `Point.py` program as follows.

```
import math
class Point(object):
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y
    def distanceTo(self, other):
        return math.hypot(self.x - other.x, self.y - other.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))
print ("p.distanceto(q) = {0}".format(p.distanceTo(q)))
```

Now run this program.

```
$ python Point.py
p = (0, 0)
q = (3, 4)
p.distanceto(q) = 5.0
$
```

The class mechanism enables us to create our own new types of objects. Python supports the class mechanism, and object-oriented programming in general.

Java goes even further: all code in Java *must* appear in a class.

3 Java Classes and Objects

A Java program consists of one or more classes. All Java code that is created is contained in classes. So far you have created a class called `Foo` containing only a main method that prints to the screen.

In Python, you created programs that consisted of functions, one of which was the "main routine," which lived outside of all other functions. Your programs had variables that point at objects and functions that remember procedures. Java has these features but it works somewhat differently. Let us begin by comparing the time–honored "Hello, World!" program in both languages. In Python we code the following

```
#!/usr/bin/python
print("Hello, World!")
```

A Java vs. Python Comparison Python has classes but their use is purely optional. In Java, all of your code *must* be enclosed in classes. Throughout you will see that Java is more "modest" than Python. No executable code can be naked; all code must occur within a function that is further clothed in a class, with the exception of certain initialization of variables that still must occur inside of a class.

Also, we should remember we have two types of statements in Python, worker statements and boss statements. In Python, boss statements are grammatically incomplete sentences. Worker statements are complete sentences. All boss statements in Python end in a colon (:), and worker statements have no mark at the end. All boss statements own a block of code consisting of one or more lines of code; an empty block can be created by using Python's *pass* keyword.

Java uses the system of boss and worker statements; the difference is cosmetic. In Java, boss statements have no mark at the end. Worker statements must be ended with a semicolon (;).

In Python, delimitation is achieved with a block structure that is shown by tabbing. In Java, delimitation is achieved by curly braces $\{\cdots\}$.

In Python, a boss statement must own a block containing at least one worker statement. In Java, a boss statement must have a block attached to it that is contained in curly braces. An empty block can be indicated by an empty pair of matching curly braces. Technically, you can get away with omitting the empty block, but it is much better to make your intent explicit.

Knowing these basic facts will make it fairly easy for you to understand simple Java programs.

Now, make the following class in Java and save it in the file `Hello.java`

```
public class Hello
{
    public void go()
    {
        System.out.println("Hello, World!");
    }
}
```

You can compile this class, but it needs a main method to run. Therefore, we add one.

```
public class Hello
{
    public void go()
    {
        System.out.println("Hello, World!");
    }
    public static void main(String[] args)
    {
        Hello greet = new Hello();
        greet.go();
    }
}
```

What's in the `main`? In the first line we see this.

```
    Hello greet  = new Hello();
```

This mysterious tells Java, "make a new object of type `Hello`." The variable `greet` points at a `Hello` object. Observe that `new` is a language keyword that is designated for creating objects. The `Hello` in front of `greet` says that `greet` is a variable of type `Hello`. Note, in contrast to Python, variables in Java

have types. In fact, the compiler requires the type of all variables be known at compile time.

On the next line, we call `h`'s `go()` method, which puts the text `Hello, World!` to `stdout`

```
$ java Hello
Hello, World!
```

You can think of the Java Virtual Machine as being an object factory. The class you make is a blueprint for the creation of objects. You may use the `new` keyword to manufacture as many objects as you wish. When you use this keyword, you tell Java what kind of object you want created, and it is brought into existence. Here we show a second `Hello` object getting created by using `new`. Modify your main method as follows

```
public static void main(String[] args)
{
    Hello greet = new Hello();
    greet.go()
    Hello snuffle = new Hello();
    snuffle.go()
}
```

Compile and run and you will see this.

```
$ javac Hello.java
$ java Hello
Hello, World!
Hello, World!
Hello@3cb075
Hello@e99ce5
```

Now there are two `Hello` objects in existence. Each has the capability to "`go()`." This is the only capability we have given our `Hello` objects so far. Every Java object has the built–in capability of representing itself as a string. The string representation of a `Hello` object looks like

```
Hello@aBunchOfHexDigits
```

You can see we have created two distinct `Hello` objects; the string representation of `greet` is `Hello@3cb075` and the string representation of `snuffle` is `Hello@e99ce5`. Each of the variables `greet` and `snuffle` is pointing at its own `Hello` object. Note that you will likely get different strings of hex digits from the ones seen here. Recall that a similar state of affairs inhered in Python;

every object is able to represent itself as a string, but the default representation is not terribly usefu.

The method `go()` represents a *behavior* of a `Hello` object. These objects have one behavior, they can `go()`. You can also see here that objects have identity. They "are". The two instances, `snuffle` and `greet` we created of the `Hello` class are different from one another. So far, we know that *objects have identity and behavior.*

This is all evocative of some things we have seen in Python. For example, if we create a string in Python, we can invoke the string method `upper()` to convert all alpha characters in the string to upper case. Here is an example of this happening.

```
>>> x = "abc123;"
>>> x.upper()
'ABC123;'
>>>
```

The Python string object and the Java Hello object behaved identically. When we sent the string `x` the message `upper()`, it returned a copy of itself with all alpha characters converted to upper-case. In the Python interactive mode, this copy is put to the Python interactive session, which acts as `stdout`.

The Java Hello object `greet` put the text `"Hello, World!"` to `stdout`.

Now let us go through the program line-by-line and explain all that we see. The first line

```
public class Hello
```

is a boss statement. Read it as "To make a Hello," Since "To make a Hello," is not a complete sentence, we know the class head is a boss statement. Therefore it gets NO semicolon.

The word `public` is an *access specifier*. In this context, it means that the class is visible outside of the file. Python has no such modesty; it lacks any system of access specifiers. Later, you may have several classes in a file. Only one may be public. You may place other classes in the same file as your public class. This is done if the other classes exist solely to serve the public class. Java requires the file bear the name of the public class in the file. The compiler will be angry if you do not do this, and you will get an error message.

The words `public` and `class` are language keywords, so do not use them as identifiers. The `class` keyword says, clearly enough, "We are making a class here." Now we go to the next line

```
{
```

This line has just an open curly brace on it. Java, in contrast to Python, has no format requirement. This freedom is dangerous. We will adopt certain formatting conventions. Use them, or develop your own and *be very consistent*. Consistent formatting makes mistakes easy to see and correct. It is unwise consistency that is the "hobgoblin of small minds;" wise consistency is a great virtue in computing.

The single curly brace acts like tabbing in in Python: it is a delimiter. It demarcates the beginning of the `Hello` class's code. The next line

```
    public void go()
```

is a function header. In Python you would write

```
def go():
```

There are some differences. The Python method header is a boss statement so it has a colon (:) at the end. Remember, boss statements in Java have no mark at the end. There is an access specifier `public` listed first. This says, "other classes can see this function." The `jshell` session behaves like a different class, so `go()` can be seen by `jshell` when we create a `Hello` object there. The other new element is the keyword `void`. A function in Java must specify the type of object it returns. If a function returns nothing, its return type *must* be marked `void`. Next you see the line

```
    {
```

This open curly brace says, "This is the beginning of the code for the function `go`." It serves as a delimiter marking the beginning of the function's body. On the next line we see the ungainly command

```
    System.out.println("Hello, World!");
```

The `System.out.println` command puts the string `"Hello, World!"` to standard output and then it adds a newline. The semicolon is required for this statement because it is a worker statement. Try removing the semicolon and recompiling. You will get a compiler error

```
> javac Hello.java
Hello.java:5: error: ';' expected
        System.out.println("Hello, World!")
                                           ^
1 error
```

A semicolon must be present at the end of all worker statements in Java. It is a mistake to put a semicolon at the end of a boss statement. In Java, the

compiler can sometimes fail to notice this and your program will have a strange logic error.

The next line

```
    }
```

signifies the end of the `go` method. Finally,

```
}
```

ends the class.

In summary, all Java code is packaged into classes. What we have seen here is that we can put functions (which we call *methods*) in classes. The methods placed in classes give objects created from classes behaviors. We shall turn next to looking at Java's type system so we can write a greater variety of methods.

Programming Exercises Add new methods to our `Hello` class with these features.

1. Have a method use `System.out.print()` twice. How is it different from `System.out.println()`?
2. Have a method do this.

   ```
   System.out.printf("<tr><td>%s</td><td>%s</td></tr>",
       16, 256);
   ```

   Experiment with this `printf` construct. Note its similarities to Python's formatting `%` construct.
3. Enter this into `jshell`.

   ```
   String s = String.format(("<tr><td>%s</td><td>%s</td></tr>",
       16, 256);
   ```

   You can, in effect "print to a String." and save yourself a lot of annoying concatenation.
4. Create the `Hello` class using Python, giving it a `go` method.

4 Java's Integer Types

Creating a new class allows you to create new types. Every time you create a class, you are actually extending the Java language. Like all things that are built out of other things, there must be some "atoms" at the bottom. Said atoms are called *primitive types* in Java. Java has eight primitive types. All other types in Java are `object types`; this includes such things as strings, lists, and graphical

widgets. Note that Python does not have this distinction; in Python these sorts of simple types are just immutable objects.

We will begin by discussing Java's four integer types. These are all primitive types. Let us begin by studying these in `jshell`. Note that the sizes are standard and do not vary with the system. This stands in contrast to C/C++, if you have studied them before.

Type	Size	Explanation
`long`	8 bytes	This is the double–wide 8 byte integer type. It stores a value between -9223372036854775808 and 9223372036854775807. These are 64-bit two's complement integers
`int`	4 bytes	This is the standard two's complement 4 byte integer type, and the most commonly used integer type. It stores a value between -2147483648 and 2147483647.
`short`	2 bytes	This is the standard 2 byte integer type. It stores a value between -32768 and 32767 in two's complement notation.
`byte`	1 byte	This is a one–byte integer that stores an integer between -128 and 127 in two's complement notation.

You should note that Python 3 has one integer type and that Python 2 has two: `int` and `long`.

Now create an integer variable named `x` and initialize it to 5 as follows.

```
jshell> int x = 5
x ==> 5
```

You are seeing the assignment operator `=` at work here. This works just as it does in Python; you should read it as, "x gets 5" and not "x equals 5." As is true in Python, it is a worker statement. Consequently in a compiled program, it must be ended with a semicolon. Observe that the `jshell` program is lenient about semicolons

The expression on the right–hand side is evaluated and stored into the variable on the left–hand side. Now enter `x` into `jshell`. It fetches the value of `x`, which is 5.

```
jshell> x
x ==> 5
```

Now enter the `/vars` command. You will see the visible symbol table.

```
jshell> /vars
|    int x = 5
```

To create a variable in Java, you need to specify its type and an identifier (variable name). This is because Java is a *statically compiled language*; the types of all variables must be known at compile time. In general a variable is created in a declaration of the form

```
type variableName;
```

You can initialize the variable when you create it like so.

```
type variableName = value;
```

When you create local variables inside of methods you should always initialize them, or the compiler will growl at you.

Now let us deliberately do something illegal. We will set a variable of type `byte` equal to 675. Watch Java rebel.

```
jshell> byte b = 675
|  Error:
|  incompatible types: possible lossy conversion from int to byte
|  byte b = 675;
|           ^-^
```

This would attract the compiler's attention in a compiled program and cause compilation to abort and for your program to error out. You should create a little class with a method doing this and see for yourself.

4.1 Using Java Integer Types in Java Code

So far we have seen Java's four integer types: `int`, `byte`, `short`, and `long`. To see them in code, begin by creating this file.

```
public class Example
{
    public void go()
    {
    }
}
```

Once you enter the code in the code window, compile and save it. It now does nothing. Now we will create some variables in the method `go` and do some experiments. Modify your code to look like this and compile.

```
public class Example
{
```

```
    public void go()
    {
        int x = 5;
        System.out.println("x = " + x);
    }
}
```

Can you make the same output using `System.out.printf`? You should try this before moving on.

Compile the program. Now open it in `jshell` using the `/open` command Then enterthe code `e = new Example()` into `jshell`. Then use the `/types` command as follows.

```
jshell> /open Example.java

jshell> /types
|    class Example

jshell> Example e = new Example();
e ==> Example@26653222

jshell>
```

Notice the mysterious item `Example@(some gibberish)`. If you noticed the gibberish looks like hex code, you are right. All Java objects can print themselves, what they print by default is not very useful. Later we will learn how to change that. Now let us send the message "`go()`" to our `Example` object `e`.

```
jshell> e.go()
x = 5
```

Recall from the `Hello` class that `System.out.println` puts things to standard output with a newline at the end.

Inside the `System.out.println()` command, we see the strange sequence `"x = " + 5`. Java has a built–in string type `String`, which is akin to Python's `str`. In Python, you would have written

```
print("x = " + str(x))
```

Java has a feature called "lazy evaluation" for strings. Once Java knows that an expression is to be a string, any other objects concatenated to the expression are automatically converted into strings and are added to the concatenation. That is why you see

```
x = 5
```

printed to `stdout`. Note that Python is very strict in this matter and requires you to explicitly convert objects to string before they can be concatenated to a string.

Now let us add some more code to our `Example` class so we can see how these integer types work together.

```
public class Example
{
    public void go()
    {
        int x = 5;
        System.out.println("x = " + x);
        byte b = x;
        System.out.println("b = " + b);
    }
}
```

Now we compile our masterpiece and we get these scoldings from Java.

```
$ javac Example.java
Example.java:7: error: incompatible types:
            possible lossy conversion from int to byte
        byte b = x;
                 ^
1 error
$
```

Indeed, line 7 contains the offending code

```
byte b = x;
```

To fully understand what is happening, let's do a quick comparison with Python and explain a few differences with Java.

Types: Java vs. Python Python is a strongly, dynamically typed language. This means that objects are aware of their type and that decisions about type are made at run time. Variables in Python are merely names; they have no type.

In contrast, Java is a strongly, statically typed language. In the symbol table, Java keeps the each variable's name, the object the variable points at and the variable's type. Types are assigned to variables at *compile time*. In Python

a variable may point at an object of any type. In Java, variables have type and may only point at objects of their own type.

Now let's return to the example. The value being pointed at by `x` is 5. This is a legitimate value for a variable of type `byte`. However, `x` is an integer variable and knows it is an integer. The variable `b` is a `byte` and it is aware of its `byte`ness. When you perform the assignment

```
b = x;
```

Java sees that `x` is an integer. An integer is a bigger variable type than a byte. The variable `b` says, "How dare you try to stuff that 4–byte integer into my one–byte capacity!" Java responds chivalrously to this plea and the compiler calls the proceedings to a halt.

In this case, you can cast a variable just as you did in Python. Modify the program as follows to cast the integer `x` to a `byte`.

```
public class Example
{
    public void go()
    {
        int x = 5;
        System.out.println("x = " + x);
        byte b = (byte) x;
        System.out.println("b = " + b);

    }
}
```

Your program will now run happily.

```
jshell> /open Example.java
jshell> Example e = new Example();
e ==> Example@26653222
jshell> e.go();
x = 5
b = 5
jshell>
```

Now let's play with fire. Change the value you assign `x` to 675.

```
public class Example
{
    public void go()
    {
```

```
        int x = 675;
        System.out.println("x = " + x);
        byte b = (byte) x;
        System.out.println("b = " + b);

    }
}
```

This compiles very happily. It runs, too!

```
jshell> Example e = new Example();
e ==> Example@26653222

jshell> e.go()
x = 675
b = -93

jshell>
```

Whoa! When casting, you can see that the doctrine of *caveat emptor* applies. If we depended upon the value of `b` for anything critical, we can see we might be headed for a nasty logic error in our code. When you cast, you are telling Java, "I know what I am doing." With that right, comes the responsibility for dealing with the consequences.

Challenge How did the -93 come about? Think about doughnutting!

Notice that you are casting from a larger type to a smaller type. This is a type of *downcasting*, and it can indeed cause errors that will leave you downcast. Since we discussed downcasting, let's look at the idea of upcasting that should easily spring to mind. For this purpose, we have created a new program that upcasts a byte to an integer

```
public class UpCast
{
    public void go()
    {
        byte b = 122;
        System.out.println("b = " + b);
        int x =  b;
        System.out.println("x = " + x);

    }
}
```

This compiles and runs without comment.

```
jshell> /open UpCast.java

jshell> UpCast u = new UpCast();
u ==> UpCast@26653222

jshell> u.go()
b = 122
x = 122
```

The four integer types are just four integers with different sizes. Be careful if casting down, as you can encounter problems. Upcasting occurs without comment. Think of this situation like a home relocation. Moving into a smaller house can be difficult. Moving into a larger one (theoretically) presents no problem with accommodating your stuff.

Important! If you use the arithmetic operators `+`, `-`, `*` or `/` on the short integral types byte and short, they are automatically upcast to integers as are their results.

Finally let us discuss the problem of type overflow and "doughnutting." Since the `byte` type is the smallest integer type, we will demonstrate these phenomena on this type. Observe that the binary operators `+`, `-`, `* /`, and `%` work in java just as they do in Python 2.x on integer types. Also we have the compound assignment operators such as += which work exactly as they do in Python.

Open `jshell` and run these commands. By saying `int b = 2147483647`, we guarantee that Java will regard `b` as a regular integer.

```
jshell> int b = 2147483647;
b ==> 2147483647

jshell> b += 1;
$5 ==> -2147483648

jshell> b
b ==> -2147483648

jshell>
```

The last command `b += 1` triggered an unexpected result. This phenomenon called *type overflow*. As you saw in the table at the beginning of the section, a byte can only hold values between -2147483648 and 2147483647. Adding 1 to 2147483647 yields -2147483648; this phenomenon is called *doughnutting*. It is an artifact of the workings of two's complement notation. You can see that this occurs in C/C++ as well.

This is caused by the fact that integers in Java are stored in two's complement notation. See the section in the Cyberdent in *Computing in Python* to learn why this happens.

5 The Rest of Java's Primitive Types

The table below shows the rest of Java's primitive types. We see there are eight primitive types, four of which are integer types.

Type	Size	Explanation
`boolean`	1 byte	This is just like Python's `bool` type. It holds a `true` or `false`. Notice that the boolean constants `true` and `false` are not capitalized as they are in Python.
`float`	4 bytes	This is an IEEE 754 floating point number It stores a value between -3.4028235E38 and 3.4028235E38.
`double`	8 bytes	This is an IEEE 754 double precision number. It stores a value between -1.7976931348623157E308 and 1.7976931348623157E308. This is the same as Python's `float` type. It is the type we will use for representing floating–point decimal numbers.
`char`	2 byte	This is a two–byte Unicode character. In contrast to Python, Java has a separate character type.

5.1 The `boolean` Type

Let us now explore booleans. Java has three boolean operations which we will show in a table

Operator	Role	Explanation
`&&`	and	This is the boolean operator $\wedge$. It is a binary infix operator and the usage is `P && Q`, where `P` and `Q` are boolean expressions. If `P` evaluates to `false`, the expression `Q` is ignored.
`||`	or	This is the boolean operator $\vee$. It is a binary infix operator and the usage is `P || Q`, where `P` and `Q` are boolean expressions. If `P` evaluates to `true`, the expression `Q` is ignored.
!	not	This negates a boolean expression. It is a unary prefix operator. Be careful to use parentheses to enforce your intent!

Hand–in–hand with booleans go the *relational operators*. These work just as they do in Python on primitive types. The operator `==` checks for equality, `!=` checks for inequality and the operators `<`, `>`, `<=` and `>=` act as expected on the various primitive types. Numbers (integer types and floating point types) have their usual orderings. Characters are ordered by their ASCII values. It is an error to use inequality comparison operators on boolean expressions.

Now let us do a little interactive session to see all this at work. You are encouraged to experiment on your own as well and to try to break things so you better understand them.

```
jshell> 5 < 7
$7 ==> true

jshell> 5 + 6 == 11   // == tests for equality
$8 ==> true

jshell> !(4 < 5)     //  ! is not
$9 ==> false

jshell> (2 < 3) && (1 + 2 == 5)  // and at work
$10 ==> false

jshell> (2 < 3) || (1 + 2 == 5)   // or at work
$11 ==> true

jshell> 100 %7 == 4    //  % is mod
$12 ==> false
```

5.2 Floating–Point Types

When dealing with floating–point numbers we will only use the `double` type. Do not test floating–point numbers for equality. Since they are stored inexactly

in memory, comparing them exactly is a dangerous hit–or–miss proposition. Instead, you can check and see if two floating–point numbers are within some tolerance of one another. Here is a little lesson for the impudent to ponder. Be chastened!

```
jshell> double q = .3
q ==> 0.3

jshell> double r = .1 + .1 + .1
r ==> 0.30000000000000004

jshell> q == r
$25 ==> false
```

All integer types will upcast to the `double` type. You can also downcast doubles to integer types; you should experiment and see what kinds of truncation occur. You should experiment with this in `jshell`. Remember, downcasting can be hazardous and . . . leave you downcast. Pay especial attention to negative numbers. If you are a number theory geek, you will have a negative reaction.

5.3 The `char` type

In Python, characters are just one–character strings. Java works differently and is more like C/C++ in this regard. It has a separate type for characters, `char`.

Recall that Western characters are really just bytes. Java uses the *unicode* scheme for encoding characters. All unicode characters are two bytes. The ASCII characters are prepended with a zero byte to make them into unicode characters. You can learn more about unicode at `http://www.unicode.org`.

Integers can be cast to characters, and the unicode value of that character will appear.

Here is a sample interactive session. Notice that the integer 945 in unicode translates into the Greek letter α.

```
> (char) 65
'A'
> (char) 97
'a'
> (char)945
'α'
> (char)946
'β'
```

Similarly, you can cast an integer to a character to determine its ASCII (or unicode) value.

The relational operators may be used on characters. Just remember that characters are ordered by their Unicode values. The numerical value for the 8 bit characters are the same in Unicode. Remember, Unicode characters are two bytes; all of the 8 bit characters begin with the byte `00000000`.

6 More Java Class Examples

Now let us develop more examples of Java classes. Since we have the primitive types in hand, we have some grist for producing useful and realistic examples. Let us recall the basics. All Java code must be enclosed in a class. So far, we have seen that classes contain methods, which behave somewhat like Python functions.

Open an editor session and place this code in a file named `MyMethods.java`.

```
public class MyMethods
{
    public double square(double x)
    {
        return x*x;
    }
}
```

Compile this program. Once it compiles, open it and `jshell` and create an instance of it.

```
jsyell> MyMethods m = new MyMethods();
```

Recall that `new` tells Java, "make a new `MyMethods` object." Furthermore, we have assigned this to the variable `m`. Now type `m.getClass()` and see `m`'s class.

```
jshell> m.getClass()
class MyMethods
```

Every Java object is born with a `getClass()` method. It behaves much like Python's `type()` function. For any object, it tells you the class that created the object. In this case, `m` is an instance of the `MyMethods` class, so `m.getClass()` returns `class MyMethods`.

We endowed our class with a `square` method; here we call it.

```
jshell> m.square(5)
25.0
```

The name of the method leaves us no surprise as to its result. Now let us look inside the method and learn its anatomy.

```
    public double square(double x)
    {
        return x*x;
    }
```

In Python, you would make this function inside of a class by doing the following.

```
class MyMethods:
    def square(self, x):
        return x*x
```

in both the top line is called the *function header*. Notice that in Python, you must use the `self` variable in the argument list for any methods you create. Python functions begin with the `def` statement; this tells Python we are defining a function. Java methods begin with an *access specifier* and then a *return type*. The access specifier controls visibility of the method. The access specifier `public` says that the `square` method is visible outside of the class `MyMethods`. The return type says that the `square` method will return a datum of type `double`.

In both Python and Java, the next thing you see is the function's name, which we have made `square`. The rules for naming methods in Java are the same as those for naming variables. To review, an identifier name may start with an alpha character or an underscore. The remaining characters may be numerals, alpha characters or underscores.

Inside the parentheses, we see different things in Java and Python. In Python, we see a lone `x`. In Java, we see `double x`. Since Java is statically typed, it requires all arguments to specify the type of the argument as well as the argument's name. This restriction is enforced *at compile time*. In contrast, Python makes these and all decisions at run time.

In general every Java method's header has the following form.

```
returnType functionName(type1 arg1, type2 arg2,  ...  , typen argn)
```

The list

```
[type1, type2, ... typen]
```

of a Java method is called the method's *signature*, or "sig" for short. Notice that the argument names are not a part of the signature of a method. Remember, such names are really just dummy placeholders. Methods in Java may have zero or more arguments, just as functions and methods do in Python.

Try entering `m.square('a')` into `jshell`.

```
jshell> m.square('a')
Error: No 'square' method in 'MyMethods' with arguments: (char)
```

Compilation would fail for this program, and , `jshell` objects by saying that a character is an illegal argument for your method `square`. Java methods have type restrictions in their arguments. Users who attempt to pass data of illegal type to these methods are rewarded with compiler errors. This sort of protection is a two–edged sword. Add this method to your `MyMethods` class.

```
    public double dublin(double x)
    {
        return x*2;
    }
```

Now let us do a little experiment.

```
jshell> /open MyMethods.java
jshell> MyMethods m = new MyMethods();
jshell> m.dublin(5)
$1 ==> 10.0
jshell> m.dublin("string")
Error: No 'dublin' method in 'MyMethods'
with arguments: (java.lang.String)
```

What have we seen? The `dublin` method belonging to the `MyMethods` class will accept integer types, which upcast to doubles, or doubles, but it rejects a string. (More about Java's string type later)

We will now write the analogous function in Python; notice what happens. Place this Python code in a file named `method.py`.

```
def dublin(x):
    return x*2
x = 5
print "dublin(" + str(x) + ") = " + str(dublin(x))
x = "string"
print "dublin(" + str(x) + ") = " + str(dublin(x))
```

Now let us run it at teh command line..

```
$ python method.py
dublin(5) = 10
dublin(string) = stringstring
$
```

Python makes decisions about objects at run time. The call `dublin(5)` is fine because it makes sense to take the number 5 and multiply it by the number 2. The call `dublin("string")` is fine for two reasons. First, multiplication of a string by an integer yields repetition, so the return statement in the function `dublin` makes sense to Python at run time. Secondly, variables in Python have no type, so there is no type restriction in `dublin`'s argument list. You will notice that static typing makes the business of methods more restrictive. However, compiler errors are better than run time errors, which can conceal ugly errors in your program's logic and which can cause surprisingly unappealing behavior from your function.

Just as in Python, you may have functions that produce no output and whose action is all side–effect. To do this, just use the `void` return type, as we did in the `Hello` class.

Programming Exercises

1. Add a method `double cube(double x)` to the class `MyMethods.java`. Can you use the `square` method to do part of the computation?
2. Modify the `square` method as follows.

   ```
   public double square(double x)
   {
       double y = x*x;
       return y;
   }
   ```

   Create a main method for this class and try placing this command in it.

   ```
   System.out.println(y);
   ```

   What happens? Why is this like Python?