Contents

0 Introduction

We are going to frame the concepts we learned in Python in Java. During this chapter, we will do a comparison of the design and mechanics of the two languages.

1 Java Data Structures

Recall that a data structure is a container in which we store a collection of related objects under a single name. Different data structures have different organizations and different rules for accessing and manipulating their contents. In Python, we met the data structures `list`, `tuple`, and `dict`. Python lists are mutable heterogeneous sequences; they can contain objects of any type as entries. Python tuples are like lists, but they are immutable; list methods that

change list state cannot be used on tuples. Python dictionaries allow us to store key-value pairs. Python data structures grow according to our needs and they shrink when we delete items from them.

Java has two data types comparable to Python lists. We begin by learning about the *array*; it is a *homogeneous* mutable sequence type of *fixed* size. When you create an array you specify its size and the type of entries it contains. If you run out of room and wish to add more entries to an array, you must create a new, bigger array, copy your array into its new home, and then abandon the old array. Before abandoning the old array, you will have do certain housekeeping chores so that all abandoned objects get garbage-collected. Arrays can be of primitive or object type. An array itself is an object. The syntax for declaring an array of type `type` is

```
> type[] identifier;
```

Open an interactive session in DrJava and reset the interactions pane. We will use our first `import` statement here. The `import` statement works much as it does in Python. Importing the class `java.util.Arrays` will give us a convenient way to print the contents of an array; the built-in string representation of an array is useless.

Let us begin by declaring a variable of integer array type.

```
> import java.util.Arrays;
> int[] x;
```

Now let's try to assign something to an entry.

```
> x[0] = 1
NullPointerException:
  at java.lang.reflect.Array.get(Native Method)
```

We are greeted by a surly error message. Here is one sure reason why.

```
> Arrays.toString(x)
"null"
```

Right now, the array variable is pointing at Java's "graveyard state" `null`. If you attempt to use a method on a object pointing at `null`, you will get a run time error called a `NullPointerException`. We need to give the array some actual memory to point to; this is where we indicate the array's size.

We call the special array constructor to attach an actual array to the array pointer `x`. After we attach the array, notice how we obtain the array's length.

```
> x = new int[10];
> Arrays.toString(x)
"[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]"
> x.length
10
```

Observe also that Java politely placed a zero in each entry in this integer array. This will happen for any primitive numerical type. If you make an array of booleans, it will be populated with the value `false`. In a character array, watch what happens.

```
> char[] y = new char[10];
> Arrays.toString(y)
"[, , , , , , , , , ]"
> (int) y[0]
0
```

The array is filled with the *null character* which has ASCII value 0. It is not a printable character. An array of object type is filled with `null`s. Typically, you will need to loop through the array to attach an object or primitive to each entry.

Arrays have indices, just as lists do in Python. Remember, you should think of the indices as living *between* the array entries. Arrays know their length, too; just use `.length`. Notice that this is NOT a method and it is an error to use parentheses at the end of it.

1.1 Goodies inside of `java.util.Arrays`

The convenience class `java.util.Arrays` is a "service class" that has useful methods for working with arrays. We will demonstrate some of its methods here. If you are working on arrays, look to it first as as a means of doing routine chores with arrays. Its methods are fast, efficient and tested. It is a nice exercise to re-create some of them, but don't needlessly reinvent the wheel.

Go to the API page; you will see some useful items there. Here is a summary of the most important ones for us. We will use `Type` to stand for a primitive or object type. Hence `Type[]` means an array of type `Type`. You call all of these methods by using `Arrays.`*method(arg(s))*.

Header	Action
`Type[] copyOf(Type[] original, int newLength)`	This copies your array and returns the copy. If the newLength is shorter than your array, your array is truncated. Otherwise, it is lengthened and the extra entries are padded with the default value of `Type`.
`Type[] copyOfRange(Type[] original, int from, int to)`	This returns a copy of a slice of your original array, between indices `from` and `to`. Using illegal entries generates a run time error.
boolean equals(Type[] array1, Type[] array2)	This returns `true` if the two arrays have the same length and contain the same values in the same order. It works just like Python's == on lists.
`void fill(Type[] array, Type value)`	This will replace all of the entries in the array `array` with the value `value`.
`void fill(Type[] array, int from, int to, Type value)`	This will replace the entries between indices `from` and `to` with the value `value`.
`String toString(Type[] array`	This pretty-prints your array as a string. You have seen this used.

1.2 Fixed Size? I'm finding this *very* confining!

We now introduce a new class and a new piece of Java syntax. An `ArrayList` is a variable-size array. There are two ways to work with `ArrayList`s and we will show them both.

Let us create an `ArrayList` and put some items in it. To work with an `ArrayList` you will need to import the class `java.util.ArrayList`. The `import` statement in the interactive session below shows how to make the class visible. `java.util.ArrayList` is the fully–qualified name of this class. The import statement puts us on a "first-name" basis with the class.

How do I know what to import? Look the class `ArrayList` up in the API guide. Near the top of the page, you will see this.

> java.util
> ArrayList<E>

This tells you that the `ArrayList` class lives in the package `java.util`. Therefore you should place this at the top of your code.

```
import java.util.ArrayList;
```

Do not put the <E> in the `import` statement.

We will use the `ArrayList`'s `add` method to place new items on the list we create.

```
> import java.util.ArrayList;
> ArrayList pool = new ArrayList();
> pool
[]
> pool.add("noodle")
true
> pool.add("chlorine")
true
> pool.add("algicide")
true
> pool
[noodle, chlorine, algicide]
> pool.get(0)
"noodle"
```

All looks pretty good here. But then there is an irritating snag.

```
> pool.get(0).charAt(0)
Error: No 'charAt' method in 'java.lang.Object' with arguments:
(int)
>
```

1.3 A Brief but Necessary Diversion: What is this `Object` object?

To explain what just happened to us properly, we will take a look into the near future that lurks in Chapter 5. Every object of object type in Java, logically enough, is an `Object`. Go into the API guide and look up the class `Object`.

Every Java class has a place in the Java class hierarchy, including the ones you create. What is different from human family trees is that a Java class has *one* parent class. A Java class can have any number of children. This hierarchy is independent of the hierarchical structure imposed on the Java Standard Library by packages.

The Java class hierarchy is an Australian (upside–down) tree, just like your file system. In LINUX, your file system has a root directory called `/`. In the Java class hierarchy, the class `Object` is the root class.

Heretofore, we have created seemingly stand–alone classes. Our classes, in fact have not really been "stand–alone." Automatically, Java enrolls them into

the class hierarchy and makes them children of the `Object` class. This is why every object has `toString()` and `equals()` methods, even if you never created them.

The only stand–alone types are the primitive types. They are entirely outside of the Java class hierarchy. However, we have seen that these too, have `Object` analogs.

What is entailed in this parent–child relationship? The child *inherits* the public portion of the parent class. In a human inheritance, the heirs can decide what to do with the property they receive. They can use the property for its original purpose or redirect it to a new purpose. In Java, the same rule applies. When we made a `toString()` method for our `Point` class, we decided to redirect our inheritance. Every Java object is born with a `toString()` method. Unfortunately the base `toString()` method gives us a default string representation of our object that looks like this.

```
ClassName@ABunchOfHexDigits
```

We decided this is not terribly useful so we *overrode* the *base* `toString()` method and replaced it with our own. To override a method in a parent class, just re–implement the method, with exactly the same signature, in the child class. We also overrode the `clone()` method in the parent class. If you intend to copy objects, do not trust the `clone()` you inherit from `Object`.

This table describes the methods in the `Object` class and the relevance of each of them to us now.

Object Method	Description
`clone()`	This method creates and returns a copy of an object. You should override this if you intend to use independent copies of instance of your class.
`finalize()`	This method is automatically called when the garbage collector arrives to reclaim the object's memory. We will rarely if ever use it.
`getClass()`	This method tells you the class that an object was created from.
`notify()`	This method is used in threaded programs. We will deal with this much later

The three `wait` methods and the `notifyAll` methods all apply in threaded programming. Threads allow our programs to spawn sub-processes that run independently of our main program. Since these are a part of `Object`, this tells you that threading is built right into the core of the Java language. We will develop threading much later.

1.4 And Now Back to the Matter at Hand

Everything is returned from an `ArrayList` is an `Object`. Strings have a `charAt()` method, but an `Object` does not. As a result you must perform a cast to use things you get from an `ArrayList`. Here is the (ugly) syntax. Ugh. It's as ugly as Scheme or Lisp.

```
> ((String)pool).get(0).charAt(0)
'n'
>
```

This is the way things were until Java 5. Now we have *generics* that allow us to specify the type of object to go in an `ArrayList`. Generics make a lot of the ugliness go away. The small price you pay is you must specify a type of object you are placing in the `ArrayList`. The type you specify is placed in the *type parameter* that goes inside the angle brackets `< .... >`. You may use any object type as a type parameter; you may not do this for primitive types.

```
ArrayList<String> farm = new ArrayList<String>();
> farm.add("cow")
true
> farm.get(0).charAt(0)
'c'
>
```

Here is something new to Java 7. Generics now have a feature called *type inference* that makes creating array lists simpler. We now show the Java 7 way to do what we just did above.

```
ArrayList<String> farm = new ArrayList<>();
> farm.add("cow")
true
> farm.get(0).charAt(0)
'c'
>
```

Notice you do not have to specify the type parameter on the right hand side.

Warning: Deception Reigns King Here! All here has a pleasing cosmetic appearance. However, it's time to take a peek behind the scenes and see the real way that generics work.

What happens behind the scenes is that the *compiler* enforces the type restriction. It also automatically inserts the needed casts for the get() method. Java then erases all evidence of generics prior to run time.

The generic mechanism should not work at run time. However, the wizards who created DrJava made generics work at run-time. You can partially blame the author of this disquisition, since he suggested it.

At run time you actually could add any type of of object to an ArrayList of strings in the interactions pane. So here is what happens behind the scenes.

1. You make an `ArrayList` of some type, say `String` by using the `ArrayList<String>` syntax.
2. You put things on the list with `add` and friends and gain access to them with the `get()` method.
3. The compiler will add the necessary casts to `String` type when you refer to the entries of the `ArrayList` using `get()`, removing this annoyance from your code.
4. The compiler then performs *type erasure*; it eliminates all mention of the type parameter from the code, so to the run time environment, `ArrayList`s look like old–style `ArrayList`s at run time.

This is a smart decision for two reasons. One reason is that it prevents legacy code from breaking. That code will get compiler growlings and warnings about "raw types" but it will continue to work.

Secondly, if you declare `ArrayList`s of various type, each type of `ArrayList` does not generate new byte code. If you are familiar with C++, you may have heard that C++'s version of generics, *templates*, causes "code bloat;" each new type declared using a C++ template creates new object code in your executable file. Because of type erasure, Java does not do this.

Let us now make a sample class that takes full advantage of generics. First, let us make a version without generics and see something go wrong.

```
import java.util.ArrayList;
public class StringList
{
    private ArrayList theList;
    public StringList()
    {
        theList = new ArrayList();
    }
    public boolean add(String newItem)
    {
        return theList.add(newItem);
    }
    public String get(int k)
    {
        return theList.get(k);
```

```
    }
}
```

Compile this program and you will get a nastygram like this.

```
1 error and 1 warning found:
-------------
*** Error ***
-------------
File: StringList.java  [line: 15]
Error: StringList.java:15: incompatible types
found   : java.lang.Object
required: java.lang.String
-------------
** Warning **
-------------
File: Java/StringList.java  [line: 11]
Warning: StringList.java:11:
 warning: [unchecked] unchecked call to add(E) as a member
of the raw type java.util.ArrayList
```

The error is that we are advertising that `get` returns a `String`; the `ArrayList`'s `get()` only returns an `Object`. Now let us add the type parameter `<String>` to the code. Your code compiles. Let us now inspect our class interactively. We can now cast aside our worries about casts.

```
> StringList greats = new StringList();
> greats.add("Babe Ruth")
true
> greats.add("Mickey Mantle")
true
> greats.add("Lou Gehrig")
true
> greats.get(0)
"Babe Ruth"
> greats.get(0).charAt(0)
'B'
>
```

You can see that `greats.get(0)` in fact returns a `String`, not just an `Object`, since it accepts the `charAt()` message.

Programming Exercises These exercises will help familiarize you with the `ArrayList` API page. This class offers an abundance of useful services. Try

these in an interactions pane session. Make sure you import `java.util.ArrayList` into the session.

1. Make a new `ArrayList` of strings named `roster`.
2. Add several lower-case words to the `ArrayList`; view its contents as you add them.
3. How do you compute the number of elements of an `ArrayList`?
4. How can you determine if a given string is in your `ArrayList`?
5. Enter this import statement: `import java.util.Collections`.
6. Type this command `Collections.sort(roster);` Tell what happens.
7. Type this command `Collections.shuffle(roster);` Tell what happens.
8. Add some upper-case words. How do they behave when you use `Collections.sort()`? What definitive conclusion can you surmise?

2 Conditional Execution in Java

Java, like Python or any other self-respecting computer language, supports conditional execution. Python has `if`, `elif` and `else` statements. These are all boss statements. All of this is the works the same way in Java, but the appearance is a little different. Here is a comparison method called `ticketTaker` in Python and Java. First we show the Python version.

```
def ticketTaker(age):
    if age < 13:
        print("You may only see G movies.")
    elif age < 17:
        print("You may only see PG or G movies.")
    elif age < 18:
        print("You may only see R, PG, or G-rated movies.")
    else:
        print("You may see any movie.")
```

The Java version is quite similar. The keywords change a bit. Notice that the predicates are enclosed in parentheses. This is required. Observe in this example that you can put a one–line statement after an `if`, `else if` or `else` without using curly braces. If you want one more than one line or an empty block attached to any of these, you must use curly braces. It is best to always use curly braces for bodies of boss statement in Java; this eliminates a lot of frustrating error messages from the compiler and a lot of irksome logic errors in your code.

```
public void ticketTaker(int age)
{
    if (age < 13)
    {
        System.out.println("You may only see G movies.");
    }
    else if (age < 17)
    {
        System.out.println("You may only see PG or G- movies.");
    }
    else if (age < 18)
    {
        System.out.println("You may only see R, PG or G movies.");
    }
    else
    {
        System.out.println("You may see any movie.");
    }
}
```

Both languages support a ternary statement. We shall illustrate it in an absolute value function for both languages. First here is the Python version.

```
def abs(x):
    return x if x >= 0 else -x
```

Now we show Java's ternary operator at work.

```
public int abs(int x)
{
    return x >= 0 ? x : -x;
}
```

Use parenthesis to keep the order of operations from producing undesired results where necessary.

Java supports an additional mechanism, the `switch` statement for conditional execution. We show an example of this statement and then explain its action.

```
public class Stand
{
    public String fruit(char c)
    {
        String out = "";
        switch(c)
```

```
        {
            case 'a': case 'A':
                out = "apple";
                break;
            case 'b': case 'B':
                out = "blueberry";
                break;
            case 'c': case 'C':
                out = "cherry";
                break;
            default:
                out = "No fruit with this letter";
        }
        return out;
    }
}
```

Let us now instantiate the `Stand` class and test its `fruit` method.

```
> s = new Stand()
Stand@6504bc
> s.fruit('A')
"apple"
> s.fruit('b')
"blueberry"
> s.fruit('z')
"No fruit with this letter"
>
```

The `switch`–`case` statement only allows you to switch on a variable of *integral type*, i.e. an integer or character type. Java 7 or later additionally allows you to switch on a `String`.

The `switch` construct cannot be used on variables of floating-point type. Clearly this is a consequence of the fact that floating-point numbers are not stored exactly and that equality comparisons between them are not at all recommended. Do not use it on a `boolean` variable; for these, we use the `if` machinery. At the end of each row of one or more `case`s, you place zero or more lines of code followed by a `break` statement. Remove various `break` statements and note the behavior of the function. You will see that they play an important role. If you do not like `switch`–`case`, you can live without it with little or no deleterious effect.

3 Extended–Precision Integer Arithmetic in Java

We shall introduce a new class, `BigInteger`, which does extended–precision integer arithmetic. Go into the Java API guide and bring up the page for `BigInteger`. Just under the main heading

> java.math
> Class BigInteger

you well see this class's family tree. Its parent is `java.lang.Number` and its grandparent is `java.lang.Object`. The fully–qualified name of the class is `java.math.BigInteger`. To use the class, you will need to put the import statement

```
import java.math.BigInteger;
```

at the top of your program. You can always look at the bottom of the family tree to see what import statement is needed.

Remember that you never need to import any class that is in `java.lang`, such as `java.lang.String`. These are automatically imported for you. Python seamlessly integrates super-long integers into the language. This is not so in Java. Java class developers cannot override the basic operators like +, -, * and /.

Begin by looking the Constructor summary. The most useful constructor to us seems to be

```
BigInteger(String val)
```

Now we shall experiment with this in an interactive session.

```
> import java.math.BigInteger;
> p = new BigInteger("1");
> p
1
>
```

We now have the number 1 stored as a `BigInteger`. Continuing our session, we attempt to compute 1 + 1.

```
> p + p
Error: Bad type in addition
>
```

In a program this would be a compiler error. Now go into the method summary and look for `add`.

```
> p.add(p)
2
> p
1
>
```

The `add` method computes `1 + 1` in `BigInteger` world and comes up with `2`. Notice that the value of `p` did not change. This is no surprise, because `BigInteger`s are immutable.

To find out if a class makes immutable objects, look in the preface on its page in the API guide. First you see the header on this page, then the family tree. Then there is a horizontal rule, and you see the text

```
public class BigInteger
extends Number
implements Comparable<BigInteger>
```

The phrase "`extends Number`" just means that the `Number` class is the parent of `BigInteger`. We will learn what "implements" means when we deal with interfaces; we do not need it now.

Next you see the preamble, which briefly describes the class. Here it says "Immutable arbitrary–precision integers." So, as with strings, you must orphan what a variable points at to get the variable to point at anything new. Now let us see exponentiation, multiplication, subtraction and division at work.

```
> import java.math.BigInteger;
> a = new BigInteger("1341121");
> BigInteger b = a.pow(5);
> a
1341121
> b
4338502129107268229778644529601
> BigInteger c = b.multiply(new BigInteger("100"))
433850212910726822977864452960100
> BigInteger d = a.subtract(new BigInteger("1121"));
> d
1340000
> d.divide(new BigInteger("1000"))
1340
>
```

It would be convenient to have a way to convert a regular integer to a big integer. There is a method

```
static BigInteger valueOf(long val)
```

To call this (static) method, the usage is

```
BigInteger.valueof(whateverIntegerYouWantConverted)
```

The `BigInteger.valueOf()` method is called a *static factory method*; it is a "factory" that converts regular integers into their bigger brethren.

We now show an example or two. Be reminded of the need to use the `equals` method when working with variables pointing at objects, so you do not get a surprise.

```
> import java.math.BigInteger;
> p  = BigInteger.valueOf(3)
3
> q = new BigInteger("3")
3
> p == q
false
> p.equals(q)
true
>
```

4 Recursion in Java

Java supports recursion, and subject to the new syntax you have learned, it works nearly the same way. All of the pitfalls and benefits you learned about in Python apply in Java. Let us write a factorial function using the `BigInteger` class Recall the structure of the factorial function in Python.

```
def factorial(n):
    return 1 if n <= 0 else n*factorial(n - 1)
```

Everything was so simple and snappy.

Now we have to convert this to Java using the operations provided by `BigInteger`. We do have some tools at hand. `BigInteger.valueOf()` converts regular integers into their bigger brethren. We also have to deal with the `.multiply` syntax to multiply. Finally, we must remember, *we are returning a* `BigInteger`. Bearing all those consideration in mind, you should get something like this. If the ternary operators is not quite to your taste, use an `if` statement instead. We have broken the big line here solely for typographical convenience.

```
import java.math.BigInteger;

public class Recursion
{
    public BigInteger factorial(int n)
    {
        return n > 0 ?
            factorial(n - 1).multiply(BigInteger.valueOf(n)):
            BigInteger.valueOf(1);
    }
}
```

Now let us test our function.

```
> r = new Recursion();
> r.factorial(6)
720
> r.factorial(100)
93326215443944152681699238856266700490715968264381621468592963895217599993229915608941463976156518286253697920827223758251185210916864000000000000000000000000
> r.factorial(1000)
40238726007709 ... (scads of digits)  ...00000
>
```

Recursion can be used as a repetition mechanism. We add a second method `repeat` to our class to character or string is passed it any specified integer number of times to imitate Python's `string * int` repeat mechanism. This will serve as a nice example of method overloading. First let us work with the `String` case. Let us call the String `s` and the integer `n`. If `n <= 0`, we should return an empty string. Otherwise, let us glue a copy of `s` to the string `repeat(s, n - 1)`

```
public String repeat(String s, int n)
{
    String out = "";
    if(n > 0)
    {
        out += s + repeat(s, n - 1);
    }
    return out;
}
```

Now we get the character case with very little work.

```
public String repeat(char ch, int n)
{
    return repeat("" + ch, n)
}
```

Now our `repeat` method will repeat a character or a string. We do not need to worry about the character or string we need to repeat. Method overloading makes sure the right method is called.

5 Looping in Java

We have already seen the `while` loop in Java. It works in a manner entirely similar to Python's `while` loop. For your convenience, here is a quick comparison

```
while predicate:
    bodyOfLoop

while(predicate)
{
    bodyOfloop
}
```

It looks pretty much the same. All of the same warnings (beware of hanging and spewing) apply for both languages. Note that the predicate of a `while` loop is enclosed in parentheses.

Java also offers a second version of the `while` loop, the `do-while` loop. Such a loop looks like this.

```
do
{
    bodyOfloop
}
while(predicate);
```

The body of the loop executes unconditionally the first time, then the predicate is checked. What is important to realize is that the predicate is checked *after* each execution of the body of the loop. When the predicate evaluates to `false`, the loop's execution ends. Almost always, you should prefer the `while` loop over the `do-while` loop. When using this loop, take note of the semicolon; you will get angry yellow if you omit it.

Java has two versions of the `for` loop. One behaves somewhat like a variant of the `while` loop and comes to us from C/C++. The other is a definite loop for iterating through a collection.

First let us look at the C/C++ `for` loop; its syntax is

```
for(initializer; test; between)
{
    loopBody
}
```

This loop works as follows. The `initializer` runs once at when the loop is first encountered. The initializer may contain variable declarations or initializations. Any variable declared here has scope only in the loop.

The `test` is a predicate. Before each repetition of the loop, the `test` is run. If the `test` fails (evaluates to `false`), the loop is done and control passes beyond the end of the loop. If the `test` passes, the code represented by `loopBody` is executed. The `between` code now executes. The `test` predicate is evaluated, if it is true, the `loopBody` executes. This process continues until the `test` fails, at which time the loop ends and control passes to the line of code immediately beyond the loop. This loop is basically a modified `while` loop.

Java also has a `for` loop for collections that works similarly to Python's `for` loop. Observe that the loop variable `k` is an iterator, just as it is in Python's `for` loop. It has a look-but-don't-touch relationship with the entries of the array, just as Python does. It grants access but does not allow mutation. This works for both class and primitive types.

```
import java.util.ArrayList;
> ArrayList<String> cats = new ArrayList<String> ();
> cats.add("siamese")
true
> cats.add("javanese")
true
> cats.add("manx")
true
> for(String k : cats){System.out.println(k);}
siamese
javanese
manx
> for(String k : cats){k = "";}//Look, but don't touch!
> for(String k : cats){System.out.println(k);}
siamese
javanese
manx
```

6 Static and `final`

You have noticed that the `static` keyword appears sometimes in the API guide. In Java, `static` means "shared." Static portions of your class are shared by all instances of the class. They must, therefore, be independent of any instance of the class, or *instance-invariant*.

When you first instantiate a class in a program, the *Java class loader* first sets up housekeeping. It loads the byte code for the class into RAM.

Before the constructor is called, any static items go in a special part of memory that is visible to all instances of the class. Think of this portion of memory as being a bulletin board visible to all instances of the class. You may make static items public or private, as you see fit. Static items that are public are visible outside and inside of the class.

When state variable or method is static, it can, and should, be called by the class's name. For instance, `BigInteger.valueOf()` is a static method that converts any `long` into a `BigInteger`. Recall we called this method a static factory method; it is static and behaves as a "factory" that accepts `long`s and converts the to `BigIntegers`.

Two other familiar examples are the `Math` and `Arrays` classes. In the `Math` class, recall you find a square-root by using `Math.sqrt()`, in `Arrays`, the static method `toString(T[])` creates a string representation of the array passed it. All of `Math`'s and `Arrays` methods are static. Neither has a public constructor. Both are called *convenience* or `service` classes that exist as containers for related methods.

You can also have variables that are declared static. In the `Math` library, there are `Math.PI` and `Math.E`. These variables are static. They are also `final`; they are immutable variables. Variables anywhere in Java can be marked `final`; this means you cannot reassign the variable once it is initialized. However, you can call mutator methods on that datum and change the state of the object a `final` variable points to. Since immutable objects and primitives lack mutator methods, these are rendered constant by declaring them `final`.

Be aware that, in this context, finality is a property of variables and not objects. What you cannot do is to make such a variable point at a different object.

If you create static variables, you should also have a `static` block in your class. Code inside this block is run when the class is first loaded. Use it to static data members. In fact, it is a desirable postcondition of your `static` block running that all static state variables are explicitly initialized. Remember *"Explicit is better than implicit"*, quoth the Zen of Python. Now let us put `final` and `static` to work.

The `Minter` class shown here gives each new instance an ID number, starting

with 1. The static variable `nextID` acts as a "well" from which ID numbers are drawn. The `IDNumber` instance variable is marked `final`, so the `ID` number cannot be changed throughout any given `Minter`'s lifetime.

```
public class Minter
{
    private static int nextID;
    final private int ID;
    static
    {
        nextID = 1;
    }
    public Minter()
    {
        ID = nextID;
        nextID++;
    }
    public String toString()
    {
        return "Minter, ID = " + ID;
    }
}
```

6.1 Etiquette Between Static and Non-Static Members

Since the Java class loader creates the static data for a class before any instance of the class is created, there is a separation between static and non–static portions of a class.

Non–static methods and state variables may access static portions of a class. This works because the static portion of the class is created before any instance of the class is created, so everything needed is in place. Outside of your class, other classes may see and use the static portions of your class that are marked `public`. These client programmers do not need to instantiate your class. They can gain access to any static class member, be it a method or a state variable by using the

```
ClassName.staticMember
```

construct.

Now consider the reverse case. Things in a class that are static must be instance-invariant. This means you cannot access the state variables or non-static methods of an object directly from a static method.

However, you can create an instance of your class and call non-static methods

on the instance. What you cannot do is have *direct* access to non-static data or methods in a class.

The key to understanding why is to know that *static data is shared by all instances of the class*. Hence, to be well-defined, *static data must be instance–invariant*. Since your class methods can, and more often than not, do depend on the state variables in your class, they in general are not instance–invariant. Static methods and variables belong to the class as a whole, not any one instance. This restriction will be enforced by the compiler. Even if a method does not depend upon a class's state, unless you declare it `static`, it is not static and static methods may not call it.

To use any class method in the non-static portion of your class, you must first instantiate the class and call the methods via that instance. We will see an example this at work in the following subsection

6.2 How do I Make my Class Executable?

To make your class executable, add the following special method.

```
public static void main(String[] args)
{
     //yourExecutableCode
}
```

To run your class, compile it. Select the interactions pane in the bottom window and hit F2. You can also type

```
> java YourClassName
```

at the prompt. Do not put any extension on `YourClassName`.

If you are on a UNIX box, you can run a Java program by entering

```
$ java YourClassName
```

at the UNIX command line.

For a simple example, place this method in the `Minter` class we just studied.

```
public static void main(String[] args)
{
    Minter m = new Minter();
    System.out.println(m);
}
```

Run the class and you will see it is now executable. Hitting the F2 button in your class's code window automatically causes the `java` command to be placed at the prompt.

```
> java Minter
Minter, IDNumber = 1
```

Observe that we made a tacit call to a method of the class `Minter`. To use the class, we had to create an instance `m` of `Minter` first. When we called `System.out.println`, we made a tacit call to `m.toString()`. You cannot make naked (no-instance) methods calls to non-static methods in `main`. You can, however, see the private parts of instances of the class.

Really, it is best to think of `main` as being outside the class and just use instance of your or other classes and use their (public) interface.

Finally, notice that `main` has an argument list with one argument, `String[] args`. This argument is an array of `String`s. This is how command–line arguments are implemented in Java. We now show a class that demonstrates this feature.

```
public class CommandLineDemo
{
    public static void main(String[] args)
    {
        int num = args.length;
        System.out.println("You entered " + num + " arguments.");
        int count = 0;

        for (String k: args)
        {
            System.out.println("args[" + count + "] = " + k);
            count ++;
        }
    }
}
```

Now we shall run our program with some command–line arguments. You need to type in the `java` command yourself rather than just hitting F2.

```
> java CommandLineDemo one two three
You entered 3 arguments.
args[0] = one
args[1] = two
args[2] = three
>
```

Even if you do not intend for your class to be executable, the `main` method is an excellent place for putting test code for your class. Making your class executable can save typing into the interactions pane. It is also necessary if you ever want to distribute your application in an *Java archive*, which is an executable file.

6.3 Running Java at the UNIX Command Line

What you need to know is how to compile a program and how to run it, if it has a `main` method at the command line. Let us use the `CommandLineDemo.java` program we just created. To compile it, enter

```
$ javac CommandLineDemo.java
```

at the command line. If you list your files with `ls`, you will see the files `CommandLineDemo.java` and `CommandLineDemo.class` in your `cwd`. Any error messages generated by the compiler will be put to `stderr`, which by default, is your terminal window. To run the program as before, use the `java` command as follows: note that the `.class` extension is not used.

```
$ java CommandLineDemo one two three
You entered 3 arguments.
args[0] = one
args[1] = two
args[2] = three
$
```

You can edit Java programs in `vi`, which affords nice syntax coloring. If you need a JDK, you can use the Oracle JDK by downloading it at the Oracle Java site, or you can obtain the `openjdk` package to enable your box for Java programming.