CS 300 for Pythonistas

Tyler Caraza-Harter and Gary Dahl

If you were introduced to programming in our CS department at UW-Madison, you probably took one of two courses: a course in the Java programming language numbered CS 200, or CS 301, a Python-based course with an emphasis on working with real datasets. The course numbering is a bit misleading, as CS 200 and CS 301 both provide an intro to programming for absolute beginners (301 is not harder despite its bigger number).

The next logical course for a student to take after one of these intro courses is CS 300 (another numbering surprise: 300 is actually more advanced than 301!). As it happens, students coming from CS 200 will have a slight advantage because CS 300, like 200, is in Java. This document is for those of you coming from CS 301 (or Pythonistas with an equivalent background).

Don't let the switch to a new programming language intimidate you. The fundamental programming logic you've learned will be the same (e.g., we'll still have types, expressions, functions, conditionals, and loops).

At the same time, there are many *syntactic* differences between the Python and Java. We've talked a bit about what syntax means in CS 301, but it's difficult to truly understand the distinction between logic and syntax until you've learned a second programming language. Once you have, you'll see many examples of how different syntax can be used to represent the same logic. For example, the first line of a while loop in Python might look something like this:

```python
while i < 10:
```

In contrast, the first line of an equivalent loop in Java looks like this:

```
while (i < 10) {
```

Logically, these loops do the same thing: they keep executing until `i` is greater than or equal to 10. But the *syntax* has two minor differences: in Java, the boolean expression, $i < 10$, must be enclosed in parentheses (Python doesn't care). In Python, the loop is followed by a colon (":"); in Java, by an opening curly brace ("{").

Relative to learning fundamental logic (e.g., how looping works in general), learning new syntax is quite easy. Of course, you must take the time to learn the syntax of Java. Until you do, programming logic that you would otherwise be quite comfortable with will appear mysterious and confusing. That's where this document comes in: we'll highlight the most striking differences between Python and Java.

1 Java Boilerplate

Do you spend more time writing code or debugging? Every programmer we know says the latter.

Now let's say you're working on a project, and you have the choice to spend 1 extra minute writing code to reduce the time you spend debugging by 2 minutes. *Would you make that tradeoff?* As it turns out, there are many ways to make such investments. One we've seen in Python is the use of `assert`. Here is an example:

```
def square_area(side):
    assert(side >= 0)
    return side ** 2
```

The above assert isn't part of the core logic of the function, but it might help you more quickly catch a bug involving negative lengths.

As a language, Java often requires you to make such investments for your own good. As a result, when writing equivalent programs in Python and Java, the code of your Java program will often be much longer. This may appall Pythonistas seeing Java code for the for the first time, but with experience, you'll come to see the merits and tradeoffs of the Java approach.

To illustrate some of the extra typing necessary, let's consider the simplest possible program: one that prints “Hello”. In Python, the program is just one line:

```
print("Hello")
```

The equivalent Java program looks like this:

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

Wow, there's a lot going on here! But don't feel overwhelmed, because you don't need to understand everything immediately. Also, with some strategic copy/paste, you won't need to type all of the above every time.

You may wish to keep around a file like the following, perhaps called `Start.java`:

```
public class Main {
    public static void main(String[] args) {
        // program logic here!
    }
}
```

Then, whenever you want to make a new program, you can copy `Start.java` to a new file, then replace the // **program logic here!** part with the code for your program. In Python, we start comments in our code with the # symbol; in Java, we instead use two forward slashes instead of the pound sign: //. Thus, the // **program logic here!** line isn't code; it's a comment for programmers.

What about the two lines before and after // **program logic here!** that you're going to be copying to start every new program? There are lots of new keywords there, such as `public`, `class`, and `static`. You'll be learning what these words mean in this class. For now, it's something that you can unthinkingly copy to every new program. Programmers often refer to this kind of mindlessly-copied code as "boilerplate code". Wikipedia describes the history of the term:

The term arose from the newspaper business. Columns and other pieces that were distributed by print syndicates were sent to subscribing newspapers in the form of prepared printing plates. Because of their resemblance to the metal plates used in the making of boilers, they became known as "boiler plates", and their resulting text - "boilerplate text". As the stories that were distributed by ready plates way were usually "fillers" rather than "serious" news, the term became synonymous with unoriginal, repetitive text.

For the most part, you don't need to think much about the code in your `Start.java` file yet. However, this is one exception. Take a look at the first line:

```java
public class Main {
```

Starting with this line of code will require you to put the code in a file named `Main.java` before you can compile/run it. If instead you wanted to put your code in a file named, for example, `Game.java`, your first line would be this:

```java
public class Game {
```

2 Dissecting Hello World

In the previous section, we introduced simple Java program. We'll now take a closer look at that program to learn three things about Java: (1) statement syntax, (2) how prints work, and (3) code-block syntax. Let's review the program:

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

Among other things, the program contains a single statement:

```
System.out.println("Hello");
```

The first thing to notice about that line is that it ends in a semicolon (";"). In Java, you need to type this at the end of every statement. This extra typing gives you more flexibility about splitting statements across multiple lines (Java doesn't care if you split a statement across multiple lines because it looks for semicolons, not newline characters). For example, although it wouldn't be advised (due to its ugliness), you could write two statements across six lines like this:

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

What are those two lines doing? Both statements are method calls. As in Python, you can call a method `fn` belonging to an object `obj` with something like this: `obj.fn()`.

For example, if you have a variable `s` referring to a string, you can split it with `s.split(" ")` in Python. In Java it's the same: `s.split(" ")`.

If you're paying close attention, you might (correctly) guess that `System.out` is an object and `println` is a method that is being called on that object.

Why is it more complicated to print in Java than in Python? The Java way is more general: `System.out` is a PrintStream object, and you can use the same `println` method on other PrintStream objects besides `System.out`. This makes it easier for you to adapt your program to send output somewhere other than the screen. There is also a `System.out.print` method you can call (which is similar, but will not produce a newline at the end of what you're printing).

The last thing we'll discuss in this section are all the "{" and "}" characters in the Java program. These characters are used to identify code blocks in your program. Consider this Python program:

```
if x < 0:
    x = -x  # line 1
print(x)  # line 2
```

How do we know that line 1 is *inside* the if-block whereas line 2 is *after* the if block? In Python, the answer is indentation: line 1 is tabbed in (making it part of the if's code block), whereas line 2 is not. Java also organizes statements into code blocks, but Java doesn't care about tabs. Instead, it considers all the code between a "{" and a "}" to be a code block. So, for example, the above Python code would correspond to the following Java:

```
if (x < 0) {
x = -x;
}
System.out.println(x);
```

Java knows that `x = -x;` is inside the conditional because it is within the if's curly braces. In contrast, the `println` statement is after the conditional because it is not inside the braces.

Although Java doesn't care about tabs, programmers generally find tabbing makes it easier to read code, so it is preferable that you write the above example as below (which is logically equivalent):

```
if (x < 0) {
    x = -x;
}
System.out.println(x);
```

To find the code block for the conditional, Java is looking for the first opening curly brace after the if. Thus, another logically equivalent (and popular) way of writing the above example is with the opening brace on its own line:

```
if (x < 0)
{
    x = -x;
}
System.out.println(x);
```

Revisiting our "hello world" example, we can now see two code blocks:

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

There is a class (we'll eventually learn what a "class" is in Java) named "Main" with a code block. That code block contains the definition of a function named "main" (lowercase). Note that instead of identifying the function definition with a `def`, as in Python, we identify the function definition with `public static void` (which will seem strange, until you learn the subtleties of different types of function definitions). The braces an the end of the function header identify the code block for the statements of this function (in this case, there is just the `println`).

In this section, we've seen major three differences between Python in Java. In each case, you must type more for Java, but there is a reward for this additional effort. In the case of `System.out.println`, we type more in trade for a more general interface; we can use the same `print` and `println` on PrintStream objects that go other places besides to the screen. We type more (a semicolon) to identify the end of a statement, allowing us to split up statements across lines more easily. Finally, we type more (curly braces) to identify code blocks. In this case, the extra effort will help you avoid bothersome semantic bugs involving mis-tabbing that are common in Python.

3 Types

Take a look at this Python program:

```
x = 3.14
```

```
print(x, type(x))
x = int(x)
print(x, type(x))
x = "pi"
print(x, type(x))
```

The output is:

```
3.14 <class 'float'>
3 <class 'int'>
pi <class 'str'>
```

Over time the value, and type, of x changes. First x refers to an int, then a float, and finally a str.

In Java, a variable's value can change, but its type cannot change. Furthermore, as the programmer, you must specify what type a variable will be used for before the variable takes a value. This is done by specifying the type before the variable name:

```
  int x;
  x = 3;
  System.out.println(x);
  x = 4;
  System.out.println(x);
  // Java would not allow us to do this because
  // 3.14 is a float, and x is an int:
  // x = 3.14;
```

By forcing you to specify the type of every variable, Java helps you avoid bugs such as the following in Python:

```
# multiple 3*4 to get 12
x = '3' # oops, we accidentally put a string in x!
y = 4
print(y * x) # with bug, this prints 3333, not 12
```

You may optionally combine the type specification and first assignment in Java:

```
  int x = 5;
```

In Java, values can either be either *primitives* or *objects* (this is a new distinction, as in Python, every value is what we might call an "object" in Java). You'll learn more about this distinction later; for now, here are some examples of primitive types and values in Java (by convention, primitive types are lower case, so "boolean" and the others are lower case):

```
boolean bo = true;       // only true or false
char    ch = 'A';        // single character
int     in = 20;         // ints between +/- 2 billion
long    lo = 300L;       // ints between +/- 10^19
float   fl = 1.2F;       // 32 bit floating point num
double  d  = 1.002;      // 64 bit floating point num
// no byte or short literals in Java, so type cast:
byte    by = (byte)2;    // 8 bit integer
short   sh = (short)-2; // 16 bit integer
```

Those last two lines are using some new syntax. In Python, type conversion is done with function calls of the form `TYPE(VALUE)`, like this:

```
x = 5.0
y = float(x)
```

In Java, you instead use `(TYPE)VALUE`. For example, we can convert a double to an int like this:

```
public class Main {
    public static void main(String[] args) {
        double x = 5.0;
        int y = (int)x;
        System.out.println(y);
    }
}
```

If instead, we had tried to just write `int y = x`, we would have gotten a an error saying this: `error: incompatible types: possible lossy conversion from double to int`.

Strings are not primitives in Java, but we'll discuss those later.

4 Operators

One of the most confusing things about switching from Python to Java is how comparisons between Strings and other objects is done. In Python, we commonly use `==` and occasionally use `is`. Let's review these:

```python
x = [1,2,3]
y = x
z = [1,2,3]
print(x == y) # True
print(x == z) # True
print(x is y) # True
print(x is z) # False
```

The variables x, y, and z all refer to a list containing the same three numbers. However, x and y are referring to the *same* list object, and z is referring to a *different* list object that happens to have the same contents. In Python, `==` (called the **equality** operator) checks whether two variables refer to two objects with the the same data; `is` (called the **identity** operator) checks whether two variables are references to the same object.

Confusingly, `==` is the identity operator in Java, making it equivalent to `is` in Python. There is no Java operator equivalent to the equality operator (`==`) in Python. Instead, you compare two objects with a call to the `.equals` method. For example:

```java
public class Main {
    public static void main(String[] args) {
        String h = "H";
        String s1 = h + "i";
        String s2 = h + "i";
        // s1 and s2 reference two different String
        // objects containing the same value, "Hi"

        // false (like Python is)
        System.out.println(s1 == s2);
        // true (like Python ==)
        System.out.println(s1.equals(s2));
    }
}
```

Just as 90% of the time you used `==` instead of `is` in Python, the vast majority of the time you're going to use `.equals` instead of `==` in Java; make a habit of it!

In Python, we've seen how we can increment or decrement a variable with the `+=` and `-=` assignment operators. For example:

```
x = 0
x += 1 # now x is 1
x -= 1 # now x is 0 again
```

These operators exist in Java too. Furthermore, incrementing or decrementing by one is so common that there's an even simpler way to do it in Java:

```
int x = 0;
x++; // now x is 1
x--; // now x is 0 again
```

An expression such as `x++` also produces a result, namely the value of x before it was incremented. For example:

```
int x = 10;
int y = x++;
// x is 11, y is 10
```

But putting the pluses before (`++x`) behaves slightly differently:

```
int x = 10;
int y = ++x;
// x is 11, and so is y
```

While there is a special power operator in Python(`**`), in Java, you'll instead use the `Math.pow` function. So instead of `2 ** 3`, you'll use `Math.pow(2, 3)`.

In Python, the logical operators are `and`, `or`, and `not`. In Java, these three operators are `&&`, `or`, and `!`, respectively. So this Python code:

```
if x > 100 and not x > 200:
  pass
```

Becomes this in Java:

```
if (x > 100 && !(x > 200)) {
  // code
}
```

5 Methods

In Python, we learned that *functions* that are associated with objects are called *methods*. In Java, we're going to call any function inside a `class` block a method. You'll eventually learned what classes are and why these definitions are actually equivalent. As it turns out, every Java function must be part of a class, so don't be surprised if people talking about Java only use the word "method" and never mention "functions."

We have already seen how we must explicitly declare the type of a Java variable. We must also explicitly state the parameter and return types of a method. We can do so something like this:

```
public static RETURNTYPE foo(PARAMTYPE x) {
  // code
}
```

Ignore the `public static` for now; just always put it there until you learn what it means. Note that the return type is specified before the method name (in this case, "foo"), and the parameter types are specified before the parameter names (in this case, we have one parameter, "x"). For example, this method takes an integer and double, and returns a String:

```
public static String foo(int param1, double param2) {
  // code
  return "result";
}
```

You need to make sure you return a result of the correct type. What if you don't want to return anything? In Python, this was not possible (without a return statement, `None` was returned for you automatically). In Java, we indicate methods that don't return anything by using the keyword `void` instead of a type, like this:

```
public static void foo(int param1, double param2) {
  // code that doesn't return anything
}
```

6 Control Flow

Control flow in Java is largely similar to control flow in Python. Conditions and while loops are similar: both use `if` and `while` behaves as expected. You will notice that `else if`, abbreviated in Python as `elif`, is not abbreviated in Java:

```
if (x > 0) {
    System.out.println(x);
} else if (x < 0) {
    System.out.println(-x);
} else {
    System.out.println("zero");
}
```

The biggest difference is that `for` loops are completely different in Java. Take a look at this loop:

```java
for (int i = 0; i < 5; i++) {
    System.out.println(i);
}
```

This snippet prints the numbers 0 through 4, each on a line. Notice in the parentheses after `for` that there are three parts, separated by semicolons. The **first part** runs before the loop, the loop keeps iterating only as long as the condition in the **second part** is true, and the **third part** runs after each iteration. You can mentally translate the for loop code to a while loop:

```java
int i = 0; // 1st part from for loop
while (i < 5) { // 2nd part from for loop
    System.out.println(i);
    i++; // 3nd part from for loop, same as i+=1
}
```

What if we want to count down over the even numbers from 10 to 0 (inclusive)? We'll want to initialize our counter to 10 (first part), loop as long as our counter is greater than or equal to 0 (second part), and subtract 2 for each iteration (third part).

```java
for (int counter = 10; counter >= 0; counter -= 2) {
    System.out.println(counter);
}
```

The above will print the numbers 10, 8, 6, 4, 2, and 0 (each on its own line).

7 Strings

The full documentation for the String API can be found here: `http://docs.oracle.com/javase/8/docs/api/java/lang/String.html`. Here's an example with some of the more commonly useful methods. It may be worth running these to compare the output to your expectations:

```
String x = "dog";
String y = "cat";
// concatenation or appending operation
System.out.println( x + y );
// compare string contents
System.out.println( x.equals(x) );
// number of chars in a string
System.out.println( x.length() );
// check whether string contains another
System.out.println( x.contains("og") );
// returns separate uppercase version of the string
System.out.println( x.toUpperCase() );
// note the string remains unchanged
// (Strings are immutable in Java, as in Python)
System.out.println( x );
// returns char at index 2 within string
System.out.println( x.charAt(2) )
// return substring starting at index 1
System.out.println( x.substring(1) );
// return substring after concatenation
System.out.println( (x+y).substring(2, 4) );
```

8 Objects

In Python, we create objects with function calls, like this:

```
l = list() # new list object
d = dict() # new dict object
```

The `list` and `dict` functions above are special; we can call them *constructors* because they construct new objects (i.e., create and initialize objects). In Python, we invoke constructors much as we invoke other functions, but in Java, we need to use the `new` keyword before invoking a constructor:

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

public class Main {
    public static void main(String[] args) {
        // an ArrayList is similar to a Python list
        ArrayList nums = new ArrayList();
        // a Hashtable is similar to a Python dict
        Hashtable mapping = new Hashtable();
    }
}
```

Note the use of `new` before the invocations. Java would complain if you were to merely write `nums = ArrayList()`.

In Java, an unassigned variable capable of referring to an object will be initialized to `null` (the equivalent of `None` in Python). If you try to access objects members using a null variable, you'll see a `java.lang.NullPointerException` Java exception.

When learning a new language, it's important to know what changes to parameters inside a method affect the arguments of the calling method. In Java, there are three things to remember: (1) if an argument is a reference to an object, the parameter will reference that same object, and so changes via the parameter will be visible after the method returns; (2) assigning a new object to a parameter has no effect outside the method; and (3) with primitives (such as ints), changes to the parameter are not visible once the method returns. Here is a complete program illustrating these three cases:

```
import java.util.ArrayList;

public class Main {
  public static void ResetArrayList1(ArrayList list) {
    // list will reference a new object
    // (an empty ArrayList); the original
    // list will not be affected by this
    list = new ArrayList();
  }

  public static void ResetArrayList2(ArrayList list) {
    // list is a reference to an object
    // (an ArrayList), so we can remove
    // all entries in the original list
    list.clear();
  }

  public static void ResetInt(int val) {
    // val is a primitive (an int),
    // so we have our own copy; this
    // has no affect outside of this method
    val = 0;
  }

  public static void main(String[] args) {
    ArrayList nums = new ArrayList();
    int x = 300;
    nums.add(300);

    // making a parameter refer to a new object
    // doesn't affect our reference
    ResetArrayList1(nums);
    System.out.println(nums.size()); // still is 1

    // however, the parameter is another reference
    // to our same object, and it can be used to
    // modify that object
    ResetArrayList2(nums);
    System.out.println(nums.size()); // now is 0

    // changes to primitives (non objects) are
    // not visible outside the method
    ResetInt(x);
    System.out.println(x); // still is 300
  }
}
```

9 Arrays

In Python, it's very common to use a list to manage a sequence of objects, and in Java, the closest thing to a Python list is the ArrayList (mentioned earlier).

There's another structure you'll frequently use in Python to manage sequences of values: the array. A Java array is less flexible than a Python list, but programs using arrays will generally run much faster. Here are some ways arrays are less flexible:

- at creation time, the type and size of an array must be specified
- the size can never change
- you can never add items of different types
- you cannot use negative indexes to access items from the end

Here is an example of variable that can be used to refer to arrays of ints:

```
int[] a;
```

Our int-array variable can refer to different arrays of different sizes at different times, even though the array objects themselves cannot change size. Array construction also requires use of the `new` keyword because arrays are objects (not primitives):

```
// create array of 3 integers (size cannot change)
int[] a = new int[3];

// we can't add new entries to that array, but we can
// reference a new (empty) array that is larger
a = new int[4];
```

Positive indexing works as expected:

```
a[0] = 300; // put 300 in first entry
System.out.println(a[0]); // print 300

// trying to access a[-1] would cause an exception
// a[-1] = 321;
```

If we want to see what's in an array, we should use the `Arrays.toString` method:

```
// printing out a reference is rarely useful
System.out.println( a );
// helper method to print array contents
System.out.println( Arrays.toString(a) );
```

Instead of checking the length of an array with a `len` function as in Python, you can instead use `.length`. Let's print every entry in our array on its own line, using a for loop and `.length`:

```
for (int i=0; i < a.length; i++) {
    System.out.println(a[i]);
}
```

What happens if we try to access entries before we've initialized them? It depends on the type; let's try it with an array of Strings.

```
public class Main {
    public static void main(String[] args) {
        String[] words = new String[3];
        for (int i=0; i < words.length; i++) {
            System.out.println(words[i]);
        }
    }
}
```

We'll see the following:

```
null
```

```
null
null
```

As you can see, the values defaulted to `null` (which is the same as `None` in Python). It's generally a good idea to put values in your arrays before you access them.

10 User Input

Scanner is an object type in the Java API that provides convenient methods for reading text from a variety of sources, including the user. It will be convenient to add the following line to the top of your java file (before: public class...) so that you can simply refer to this type as Scanner instead of java.util.Scanner throughout the file.

```java
import java.util.Scanner;
```

The first line of the following example creates a single scanner to read input from the user through standard in. (*Note that you should never create more than one of these, because the resulting behavior is not defined in the Java specification and is likely to differ in different run-time environments.*) Once this scanner object has been created, you can use methods like nextInt(), nextDouble(), next(), and nextLine() to read successive input from the user. All of the methods in this class are documented here: `http://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html`. The close() method at the end signifies that we are done reading data through this scanner object.

```java
Scanner in = new Scanner(System.in);
System.out.print("Please enter your age: ");
int age = in.nextInt();
System.out.print("Please enter your name: ");
String name1 = in.nextLine();
String name2 = in.nextLine();
System.out.println("("+age+","+name1+","+name2+")");
in.close();
```

Note there is an important and often confusing detail about the difference between the Scanner's nextLine() method versus the other next(), nextInt(), nextDouble(), etc. methods. nextLine() always returns a string with all (or any) unread characters including whitespace through the end of the current line. The other methods skip over any initial whitespace before starting to read a value, and then read in and return the next value that they are able to read of the appropriate type.

The above example demonstrates a case that is often confusing for students new to Java and the details of how this Scanner class works. Specifically, after the user types their age and presses enter, nextInt() reads and returns their age. The confusing part is that nextLine() also runs, reads in the newline/enter that the user pressed after their age, and returns an empty string since no other characters were left to be read in before the end of that line. The next line that the user enters will be read in by the second nextLine() and stored in the name2 variable. It could be worth experimenting with variations of this example to help ensure that you are clear on the behavior of these Scanner methods.