**Introduction**

We will use Python as a starting point for our journey into [Java](https://en.wikipedia.org/wiki/Java_(programming_language)). We will begin by looking at a very simple Java program, just to see what the language looks like and how we get a program to run. Next, we will look at the main constructs that are common to most programming languages. These concepts are part of what is generally referred to as [procedural programming](https://en.wikipedia.org/wiki/Procedural_programming).

* Data Types
* Loops
* Reading user input
* Conditionals

Once we have the basics of Java behind us we will move on to look at the features of Java that are both unique and powerful. These features bring us into the areas of [object-oriented programming](https://en.wikipedia.org/wiki/Object-oriented_programming) and [data structures](https://en.wikipedia.org/wiki/Data_structure).

* Classes
* Interfaces
* Abstracts
* Collections

**Why Learn Another Programming Language?**

Python is great first programming language for several reasons. The syntax is sparse and clear. The underlying model of how objects and variables work is very consistent. You can write powerful and interesting programs without a lot of work. However, Python is representative of one kind of language, called a dynamic language. You might think of Python as being fairly informal. There are other languages, like Java and C++ that are more formal.

These languages have some advantages of their own. For very large programs, languages like Java and C++ are going to give you the best performance. They are also much more "safe" and "maintainable". A lot of what makes Python easy to use is that you must remember certain things. For example, if you set a variable x to reference a turtle, and forget later that x is a turtle but try to invoke a string method on it, you will get an error upon running your program. Java and C++ protect you by forcing you to be upfront and formal about the kind of object each variable is going to refer to. This means that some types of errors are caught sooner, and the runtime behavior of a program is more dependable.

In one sense, Python is representative of a whole class of languages, sometimes referred to as "scripting languages." Other languages in the same category as Python are Ruby and Javascript. Java is representative of what I will call industrial strength languages. These languages are good for projects with several people working on the project where being formal and careful about what you do may impact lots of other people. Languages in this category include C++, C, and C#.

Programming languages will always change. As the field of computer science advances there will be new programming languages, and you will need to learn some of them. As you do, you will see that there are certain features that most programming languages have in common - variables, loops, conditionals, functions - and there are some features that are unique to each. If you know what is common in languages, that is a good place to start.

**Why Learn Java?**

Java has been widely used by programmers for over 20 years. It continues to grow and to evolve, and to be extremely relevant to both programmers and companies using and building software. At a high level, this can easily be seen by looking at measures of language popularity and usage, such as the [Tiobe Index](http://www.tiobe.com/tiobe-index/) or [GitHut 2.0](https://madnight.github.io/githut/). In addition, Java is a [C-style language](https://en.wikipedia.org/wiki/Category:C_programming_language_family), meaning that many of its syntax features are derivative of the [C programming language](https://en.wikipedia.org/wiki/C_(programming_language)). This makes learning the basics of other C-style languages relatively straightforward once you know Java (or vice versa).

Java is an enormous language. There are over 4200 different classes included in the Java 8 Standard Edition. We could not begin to scratch the surface of these classes even if we devoted all of our time to the task. However, Java is very powerful and we will write some very powerful programs in this course.

**Getting Ready**

Before starting with **each** of these lessons, open up IntelliJ and using the Project pane browse to the java4python package under org.launchcode.java.demos in the src folder. All Java examples in these lessons are provided in this package, and you should run the programs there as you go, modifying and experimenting with the code to help you learn.

When looking at source code for these examples in IntelliJ, you will notice that they vary slightly from code included here. In these lessons, we omit package declarations, top-level comments, and other inconsequential elements.

We'll use Python 3 in each of our Python examples.

**Hello, World**

As with learning Python, we'll start by writing a "Hello, World" program. There are no logic errors to make, so getting it to run relies only on understanding the most basic syntax rules of the language.

**Comments**

Before looking at our "Hello, World" program, let's start by covering the easiest piece of syntax in any programming language: comments.

In Python we were able to insert comments in two ways. Single line comments begin with # and multi-line comments (aka docstrings) are surrounded by triple-quotes, """.

# This comment lives on a single line

"""this comment

can span

multiple lines

"""

We have similar functionality in Java, but with // denoting single-line comments and /\* and \*/ surrounding multi-line comments.

// This comment lives on a single line

/\*

this comment

can span

multiple lines

\*/

Throughout these initial lessons, we'll use a short comment at the top right of each code sample to indicate which language is being used. Here's an example:

...some Java code...

**Hello, World in Python**

Let's first look at a version of Hello, World for Python:

print("Hello, World")

If you put this code in a file named hello-world.py, you could run it within a terminal by issuing the command python hello-world.py. It would then print Hello, World to the terminal.

Let's take this example one step further, to draw a closer parallel to what we'll see in Java. Consider the following, more complicated program.

def main():

print("Hello, World")

if \_\_name\_\_ == '\_\_main\_\_':

main()

Here, we've added a "wrapper" to our simple call to the print function, in the form of a main function. Recall that the last two lines of the program check to ensure that the file is being run directly, rather than being imported as a module into another Python file. Running the program with python hello-world.py results in the same output as the first example.

**Hello, World in Java**

In IntelliJ you'll see a Java program named Hello.java with the same functionality as the Python Hello, World program:

public class Hello {

public static void main(String[] args) {

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

}

}

Comparing this to the Python version above, the only real similarities we see are the "Hello, World" string and functions that seem to indicate they print a message (print and System.out.println). However, there is more that's different than the same. Do not worry! An important skill for a programmer is to learn what to ignore and what to look at carefully. You will soon find that there are some elements of Java that will fade into the background as you become used to seeing them.

This simple example illustrates a few very important Java rules:

1. Every Java program must define a class; all code is inside a class
2. Everything in Java must have a type
3. Every Java program must have a function (or method) called main that has a signature\* of public static void main(String[] args)

\* A **method signature** specifies all of the information necessary for a programmer to use the method, including, at minimum, its name and the number, types, and order of its parameters. It can also include access and static modifiers (we'll learn about these later).

Let's take this example one line at a time to see how these rules are applied.

**Line 1**

Our program starts with public class Hello, along with an opening {. The first word, public, indicates to the Java compiler that this is a method that anyone can call. We will see soon that Java enforces several levels of security on the methods we write, including **public**, **protected**, and **private** methods.

The class keyword indicates that we are defining a class, while Hello is the name of the class. The name of a class is up to the programmer, and can be anything that fits within some basic rules (names must start with a letter, may not contain spaces, and so on). When naming classes it's best to follow [Java naming conventions](https://education.launchcode.org/skills-back-end-java/java4python/naming-conventions/).

Unlike Python, where a program can simply be a bunch of statements in a file, Java programs must be inside a class. The Hello class is not a very useful class since it has no instance variables and only one method. It is created out of necessity.

When defining a class, we must surround the contents of the class (its properties and methods) with "curly braces": { and }. These braces will also be used to surround conditionals, methods, and loops in coming lessons. In Java, any section of code enclosed in a pair of such braces is referred to as a **block**.

**Line 2**

On the next line we start our method definition. The name of this method is:

public static void main(String[] args)

You will notice that we indented this line once, as it sits one level inside of our class definition. Unlike in Python, such indentation is not required; it is simply a convention applied for the purpose of readability and consistency. Instead of indentation, what signifies to the compiler that this method definition is part of the Hello class is the fact that it is part of the block associated with that class. In other words, it's the outer set of curly braces that make this explicit.

Everything on this line is significant, and is part of the identification of this method. For example, the following lines look similar but are in fact treated by Java as completely different methods:

* public void main(String[] args)
* public static void main(String args)
* public static void main()
* void main(String args)

Dissecting this one line could easily take us very deep into the world of Java, but we'll be cautious to not get pulled to far in too soon. Each of these concepts can and will be explored in much more depth later.

**public**

Similar to the use of public in line 1, this indicates to the Java compiler that this is a method that anyone can call.

**static**

The next word, static, tells Java that this is a method that is part of the class, but does not belong to any one instance of the class. Rather, it is shared by all instances. Recall that a class is a blueprint for a type of object, and when we create an object based on a class, we refer to that object as an **instance** of the class (or just **instance**).

The kind of methods we typically wrote in Python required an instance in order for the method to be called. With a static method, the object to the left of the . is a class, not an instance of the class. The way that we would call the main method directly is: Hello.main(parameter1). For now you can think of static methods the same way you think of methods in Python modules that don’t require an instance. For example, the math module contains many methods: sin, cos, etc. You evaluated these methods using the names math.cos(90) or math.sin(60).

**void**

The next word, void, tells the Java compiler that the method main will not return a value. This is roughly analogous to omitting the return statement in a Python method. The method will run to completion and exit but will not return a value that you might use in an assignment statement. As we look at other examples we will see that every Java function must tell the compiler what kind of an object it will return. This is in keeping with the rule that says everything in Java must have a type. In this case we use the special type called void which means nothing will be returned.

**main**

Next we have the proper name for the method: main. The rules for names in Java are similar to the rules in Python. Names can include letters, numbers, and the \_. Names in Java must start with a letter.

**String[] args**

Finally, we have the parameter list for the method. In this example we have one parameter, named args. Everything in Java must have a type, so we also have to tell the compiler that the value of args is an array of strings. For the moment you can just think of an array as being the same thing as a list in Python. The practical benefit of declaring that the method main must accept one parameter and the parameter must be a an array of strings is that if you call main somewhere else in your code and pass it an array of integers or even a single string, the compiler will flag it as an error. In Python we would not have seen such an error manifest itself until we ran the program.

**Line 3**

Let's look at the next line: System.out.println("Hello, World");. This line should look a bit more familiar to you. Python and Java both use the dot notation for finding names.

In this example we start with System. System is a class. Within the System class we find the object named out. The out object is the standard output stream for this program (typically, the command prompt). Having located the out object Java will now call the method named println(String s) on that object. The println method prints a string and adds a newline character at the end. Anywhere in Python that you used the print function you will use the System.out.println method in Java.

**Semi-colon**

There is one more character on this line that is significant and that is the ; at the end. In Java, the ; signifies the end of a statement. Unlike Python where statements are almost always only one line long, Java statements can spread across many lines. The compiler knows it has reached the end of a statement when it encounters a ;. This is a very important difference to remember. In Java the following statements are all legal and equivalent. We do not encourage you to write your code in any way other than the first example, but you should know that it is legal.

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

System.out.println("Hello, World")

;

System.out.println

(

"Hello, World"

) ;

System.

out.

println("Hello, World")

;

In contrast to Python, indentation (and whitespace, in general) *does not* hold any special meaning in Java.

**Lines 4-5**

The last two lines of the Hello, World program simply close the two blocks. The first, or "outer", block is the class definition. The second, or "inner", block is the function definition.

If we wanted to translate the Java back to Python, we would have something like the following class definition:

class Hello(object):

@staticmethod

def main(args):

print "Hello, World"

Notice that we used the decorator @staticmethod to tell the Python interpreter that main is going to be a static method. The impact of this is that we don’t have to - and indeed we should not - use self as the first parameter of the main method! Using this definition we can call the main method in a Python session like this:

>>> Hello.main("")

Hello, World

>>>

**References**

* [Java (programming language) (wikipedia.org)](https://en.wikipedia.org/wiki/Java_(programming_language))
* [The History of Java Technology (oracle.com)](http://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html)

**Data Types**

How Java handles values and variables is extremely different from how they are implemented in Python.

**Static vs. Dynamic Typing**

Python is a **dynamically typed** language. In a dynamically typed language a variable or parameter can refer to any kind of value at any time. When the variable is used, the interpreter figures out what type it is and behaves accordingly.

Java is a **statically typed** language. In a statically typed language the association between a variable or parameter and the type of value it can refer to is determined *when the variable or parameter is declared*. Once the declaration is made it is illegal for it to refer to a value of any other type.

For example, this is legal in Python:

x = "dog"

x = 42

If we were to inspect the type of x after the first line executes (e.g. using Python's type() function) we would find that it was a string. After the next line executes, it is an integer. x is allowed to hold values of different types.

However, the corresponding code in Java will result in a compiler error:

String x = "dog";

x = 42;

The compiler error would occur when we try to assign 42 to a variable of type String. This is simply not allowed in Java.

Formally, this means that *we must declare the type of every variable and parameter* in a statically typed language. This is done by preceding the variable or parameter name with the name of its type, as we did in the example above: String x = "dog".

We only need to specify the type of a variable or parameter when declaring it. Subsequent usage does not require specifying the type, and will result in an error.

Dynamic and static typing are examples of different [type systems](https://en.wikipedia.org/wiki/Type_system). The type system of a programming language is one of the most important high-level characteristics that programmers use when discussing the differences between languages. Here are a few examples of popular languages falling into these two categories:

* **Dynamic**: Python, Ruby, Javascript, PHP
* **Static**: Java, C, C++, C#, Ada

This major difference between Python and Java will mean that we'll need to pay much more attention to types when writing Java. Let's begin by exploring the most common and important data types in this language.

**String**

Strings in Java and Python are quite similar. Like Python, Java strings are immutable. However, manipulating strings in Java is not quite as obvious since strings do not support an indexing or slicing operator. That is not to say that you can’t index into a Java string; you can. You can also pull out a substring just as you can with slicing. The difference is that Java uses method calls where Python uses operators.

This is the first example of another big difference between Java and Python: Java does not support operator overloading. The table below maps common Python string operations to their Java counterparts. For the examples shown in the table, we will use a string variable called str.

| **Python** | **Java** | **Description** |
| --- | --- | --- |
| str[3] | str.charAt(3) | Return character in 3rd position |
| str[2:5] | str.substring(2,4) | Return substring from 2nd to 4th |
| len(str) | str.length() | Return the length of the string |
| str.find('x') | str.indexOf('x') | Find the first occurrence of 'x' |
| str.split() | str.split('\s') | Split the string on whitespace into a list/array of strings |
| str.split(',') | str.split(',') | Split the string at ',' into a list/array of strings |
| str + str | str.concat(str) | Concatenate two strings together |
| str.strip() | str.trim() | Remove any whitespace at the beginning or end |
| str.lower() | str.toLowerCase() | Make all alphabetic characters in the string lower case |
| str.upper() | str.toUpperCase() | Make all alphabetic characters in the string upper case |
| 'text' in str | str.contains('text') | Search for character sequence within a string, returns a boolean |
| str == 'text' | str.equals('text') | Compares strings for equality and returns a boolean. (In Java, using == will just compare the references). |

**Primitive Types**

One of the great things about Python is that all of the basic data types are objects. Integers are objects, floating point numbers are objects, lists are objects, everything is an object. In Java that is not the case; some of the most basic data types like integers and floating point numbers are not objects. The benefit of having these **primitive** data types be non-objects is that operations on the primitives are fast. Historically, however, it became difficult for programmers to combine objects and non-objects in the way that we do in Python. So as Java evolved, eventually all the non-object primitives ended up with objectified versions, commonly called *wrapper classes*.

| **Primitive** | **Wrapper Class** | **Examples** | **Notes** |
| --- | --- | --- | --- |
| int | Integer | -5  1024 |  |
| float | Float | 1.212  3.14 |  |
| double | Double | 3.14159  2.0 | Doubles are twice as precise (i.e. can hold much longer decimal numbers) than floats |
| char | Char | 'a'  '!' | A single Unicode character. Must be enclosed in single quotes '' to be a character; double-quotes "" indicate a string |
| boolean | Boolean | true  false | Note that booleans in Java are not capitalized as they are in Python. |

Not all primitive types in Java are listed here, only the most commonly used types that beginners are likely to encounter. If you're curious, [read more about primitive types in Java](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html)

**Autoboxing**

In older versions of Java it was the programmer's responsibility to convert back and forth from a primitive to an object whenever necessary. This involved converting a value of a primitive type to an object type, or vice versa. It looked like this:

int x = 5;

Integer y = Integer.valueOf(x);

int z = (int) y;

This processing of converting a primitive to an object (e.g. Integer y = Integer.valueOf(x)) was called **boxing**. The reverse process (e.g. int z = (int) y) is called **unboxing**. In Java 5, the compiler became smart enough to know when to convert back and forth. This process is called **autoboxing**. The consequence of autoboxing for the Java programmer is that in many situations you can use primitive and object types interchangeably.

It's a best practice to use primitives whenever possible. The primary exception to this occurs when storing values in collections, which we'll learn about in a future lesson.

**Example: The TempConv Program**

Let's go back in time and look at another of our very early Python programs. Here is a simple Python function to convert a Fahrenheit temperature to Celsius.

def main():

fahrenheit = int(input("Enter the temperature in F: "))

celsius = (fahrenheit - 32) \* (5.0 / 9.0)

print("the temperature in C is: ", celsius)

if \_\_name\_\_ == '\_\_main\_\_':

main()

Next, lets look at the Java equivalent, which is in the java4python folder in IntelliJ

import java.util.Scanner;

public class TempConv {

public static void main(String[] args) {

double fahrenheit;

double celsius;

Scanner in;

in = new Scanner(System.in);

System.out.println("Enter the temperature in F: ");

fahrenheit = in.nextDouble();

celsius = (fahrenheit - 32) \* 5.0/9.0;

System.out.println("The temperature in C is: " + celsius);

}

}

There are several new concepts introduced in this example. We will look at them in the following order:

* import statement
* Variable declaration
* Input/output and the Scanner class

**import**

In Java, you can use any class that is available without having to import the class, but you must adhere to two very important conditions:

1. The javac and java commands must know that the class exists.
2. You must use the full name of the class

How do the java and javac commands know that certain classes exist? We have these rules:

1. Java knows about all the classes that are defined in .java and .class files in your current working directory.
2. Java knows about all the classes that are shipped with java.
3. Java knows about all the classes that are included in your CLASSPATH environment variable. Your CLASSPATH environment variable can name two kinds of structures:
   1. A jar file that contains java classes. (A jar file is a "Java archive", and ends in .jar. For now, think of it as a zip file that contains a bunch of classes.)
   2. Another Unix directory that contains Java class files.

You can think of the import statement in Java as working a little bit like the from module import xxx statement in Python. However, behind the scenes the two statements actually do very different things.

The first important difference to understand is that the class naming system in Java is very hierarchical. The *full* name of the Scanner class is really java.util.Scanner. You can think of this name as having two parts: The first part java.util is called the **package**, and the last part is the class. We’ll talk more about the class naming system a bit later. The second important difference is that it is the Java class loader’s responsibility to load classes into memory, not the import statement’s.

The import statement tells the compiler that we are going to use a shortened version of the class’s name. In this example we are going to use the class java.util.Scanner, but we can refer to it as just Scanner. We could use the java.util.Scanner class without any problem and without any import statement provided that we always referred to it by its full name.

Don't just trust us, try it yourself! Remove the import statement and change Scanner to java.util.Scanner in the rest of the code. The program should still compile and run.

**Declaring Variables**

In the example above, these lines contain variable declarations:

double fahrenheit;

double celsius;

Scanner in;

Specifically we are saying that fahrenheit and celsius are going to reference objects that are of type double. This means that if we were to try an assignment like fahrenheit = "xyz" the compiler would generate an error because "xyz" is a string and fahrenheit is supposed to be a double. The variable in will reference a Scanner object.

For Python programmers the following error is likely to be even more common. Suppose we forgot the declaration for celsius. What would happen if we try to manually compile our programing using javac TempConv.java on the command line?

TempConv.java:13: cannot find symbol

symbol : variable celsius

location: class TempConv

celsius = (fahrenheit - 32) \* 5.0/9.0;

^

TempConv.java:14: cannot find symbol

symbol : variable celsius

location: class TempConv

System.out.println("The temperature in C is: " + celsius);

^

2 errors

When you see the first kind of error, where the ^ symbol is on the left side of the assignment operator, it usually means that you have not declared the variable. If you have ever tried to use a Python variable that you have not initialized the second error message will be familiar to you. The difference here is that we see the message before we ever try to test our program.

When using an IDE such as IntelliJ, your code is typically checked by the IDE's built-in compiler as you write your code. Thus, errors are usually visually indicated within your code by the IDE as you write your code, saving you the extra step of having to explicitly compile your code before finding compiler errors. Nice, huh?

The general rule in Java is that you must decide what kind of an object your variable is going to reference and then you must declare that variable before you use it. There is much more to say about the static typing of Java but for now this is enough.

**Input / Output and the Scanner Class**

In the previous section we created a Scanner object. In Java, Scanner objects make getting input from the user, a file, or even over the network relatively easy. In our case we simply want to ask the user to type in a number at the command line, so we construct a Scanner instance by using the word new and then calling the constructor and passing it the System.in object:

in = new Scanner(System.in);

Notice that this Scanner object is assigned to the name in, which we declared to be a Scanner earlier in the program. System.in is similar to System.out except of course it is used for input. If you are wondering why we must create a Scanner to read data from System.in when we can write data directly to System.out using println, you are not alone. We will talk about the reasons why this is so later when we talk in depth about Java streams. You will also see in other examples that we can create a Scanner by passing the Scanner a File object. You can think of a Scanner as a kind of "adapter" that makes low level objects easier to use.

As in Python, in Java you may declare and initialize your variables in the same line:

Scanner in = new Scanner(System.in);

On this line we use the Scanner object to read in a number:

fahrenheit = in.nextDouble();

Here again we see the implications of Java being a strongly typed language. Notice that we must call the method nextDouble, because the variable fahrenheit was declared as a double.

As a consequence of Java's type system, Scanner must have a function that is guaranteed to return each kind of object it wants to be able to read. The compiler matches up these assignment statements and if you try to assign the results of a method call to the wrong kind of variable it will be flagged as an error.

The table below shows you some commonly used methods of the scanner class. There are many more methods supported by this class and we will talk about how to find them in the next chapter.

| **Return type** | **Method name** | **Description** |
| --- | --- | --- |
| boolean | hasNext() | returns true if more data is present |
| boolean | hasNextInt() | returns true if the next thing to read is an integer |
| boolean | hasNextFloat() | returns true if the next thing to read is a float |
| boolean | hasNextDouble() | returns true if the next thing to read is a double |
| Integer | nextInt() | returns the next thing to read as an Integer |
| Float | nextFloat() | returns the next thing to read as a Float |
| Double | nextDouble() | returns the next thing to read as a Double |
| String | next() | returns the next thing to read as a String |
| String | nextLine() | returns the next line read as a String |

**Class Types**

In addition to the types introduced so far - primitives and their objectified counterparts - any class in Java defines a type. Classes and objects are conceptually the same as in Python: A class is a template for creating objects. We'll have much more to say about classes and objects, but for now you need to be comfortable seeing the basic syntax of class types and class creation.

If I have a class Cat with a constructor that takes no arguments, I can declare and create a new instance of Cat using its constructor. In Python, we did this as follows:

# Python

my\_cat = Cat()

And the Java version is:

Cat myCat = new Cat();

Each of these statements creates a new variable that is initialized to hold a new Cat object. Note that in Java, we must declare the variable's type. Also note that we precede the constructor with the new keyword. And, of course, the Java example ends with a semi-colon.

Variables and parameters that are of the type of a class are said to be of **reference type** (in contrast to **primitive type**). In plain English, we would say of the Java example: "myCat is a reference variable of type Cat."

**References**

Reference types are different from primitive types in an essential way. A variable of a reference type (such as myCat above) does not actually store the object in question. Instead, it stores a **reference** to the object. A reference is literally a memory address. We visualize references as an arrow pointing to the object in memory.

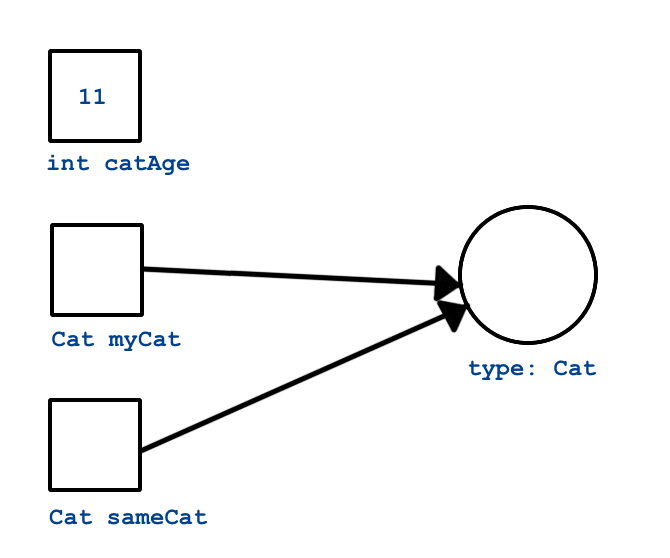
Consider this code:

int catAge = 11;

Cat myCat = new Cat();

Cat sameCat = myCat;

Visually, we can represent these three variables as shown below.



Since int is a primitive type, the variable catAge functions as a box holding the integer value 11. On the other hand, myCat is a reference variable, since it holds an object. The variable actually stores the address of the object, which we visualize as an arrow from the variable box to the object.

When we assign a value to a reference type, as in Cat sameCat = myCat, we are not creating a second copy of the object, but instead are creating a second "arrow" pointing to the same object.

The distinction between references types and primitives is important, if difficult, to wrap your brain around at first. We will see that reference types are handled differently in essential and important ways in a lot of different situations.

**Arrays**

Just as Java has primitive types for things that were objects in Python -- such as integers and booleans -- it also has a type that you might consider to be a "primitive list", arrays.

An array is an ordered, fixed-size collection of elements. Since Java is statically-typed, arrays may only store one type of object. We can create an array of integers or an array of strings, but we may not create an array that holds both integers and strings.

The syntax for creating an array capable of holding 10 integers is:

int[] someInts = new int[10];

To create an array of a different size, replace the number 10 in brackets with the desired size. To create an array holding a different type, replace int (on both sides of the assignment) with the desired type, for instance, double. Unlike lists in Python, arrays in Java *may not* change size once created. This turns out to be not very practical, so thankfully Java provides more flexible ways to store data, which we'll explore in a later lesson.

In addition to the technique above, we can initialize an array using a literal expression:

int[] someOtherInts = {1, 1, 2, 3, 5, 8};

Here, the size is implicit in the number of elements in the literal expression {1, 1, 2, 3, 5, 8}.

To access array elements, we use square brackets, as with Python lists.

int anInt = someInts[0];

As with Python lists, arrays have *zero-based indexing*.

Aside from using arrays to build some simple loop examples in the next lesson, we'll only use them in special cases. However, they're a core part of Java, so it's good to know how they work.

**Static Methods**

In pure object-oriented languages like Java and C#, we don't have functions in the sense you're used to. For example, functions may not be declared outside of a class. Within the context of a class, functions are referred to as **methods**. We will adopt this terminology from now on, and you are encouraged not to refer to methods as "functions" when you are talking about Java code.

We'll dive into learning about classes and objects in Java soon enough, but until we do, we'll frequently need to write methods, so we should understand a little bit about them. In particular, we'll use **static methods**, which behave similarly to functions as you knew them in Python.

A static method is one with the static keyword, as our main method above has:

public static void main(String[] args)

{

// some code

}

We've already explored each element of this line, however, we haven't really shown you how you might create your own methods in other contexts. To do so involves using a different name for our method, and adjusting the return type and parameter types accordingly.

Until we get into Object Oriented Programming, every method we write will have the static keyword. Leaving off static will prevent you from calling methods you define as you would like to.

We will explore exactly what static does in more detail in later lessons.

Let's create two classes in Java to demonstrate this. One will have a main method and the other will have a method that we want to call from within main.

public class HelloMethods {

public static void main(String[] args) {

String message = Message.getMessage("fr");

System.out.println(message);

}

}

public class Message {

public static String getMessage(String lang) {

if (lang.equals("sp")) {

return "Hola Mundo";

} else if (lang.equals("fr")) {

return "Bonjour le monde";

} else {

return "Hello World";

}

}

}

We won't explore every new aspect of this example, but rather will focus on the two methods.

As you've been following along with these examples using the code in IntelliJ, you've probably noticed that each class file, for example Message.java and HelloMethods.java, is named exactly the same as the class that file holds, for example Message and HelloMethods respectively.

It is a rule in Java that a file containing a class marked public **must** be named the same as that class. So remember to name each Java file you create to match the public class that file contains.

The main method in the HelloMethods class is the same in structure as that of our previous examples. Take a look at the Message class. Note that it *does not* have a main method, so it can't be run on it's own. Code within the Message class must be called from elsewhere in order to execute.

The Message class has a method of its own: getMessage. Like main, it has the static keyword. Unlike main, getMessage has a return type of String. It also has a single parameter, String lang.

Since Java is statically typed, each method must declare its return type -- that is, the data type of what it will return -- along with the type of each parameter. One consequence of this that may not be immediately obvious is that a method in Java may not have return statements that return different types of data. For example, we would not be able to replace the last return statement of getMessage with something like return 42;. This would be flagged as a compiler error.

Finally, let's note how a static method is called. The first line of main in the HelloMethods class is:

Message.getMessage("fr");

To call a static method we must use the name of the class in which it is defined, followed by ., followed by the name of the method.

We are able to call this method from another class because it is declared to be public. If we wanted to restrict the method from being called by another class, we could instead use the private modifier. We'll explore access modifiers in more depth in coming lessons.

**References**

* [Primitive Data Types (docs.oracle.com)](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html)
* [Autoboxing and Unboxing (docs.oracle.com)](http://docs.oracle.com/javase/tutorial/java/data/autoboxing.html)
* [Variables (docs.oracle.com)](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/variables.html)
* [Arrays (docs.oracle.com)](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html)

**Control Flow**

In this section we examine the syntax of control flow statements in Java - conditionals and loops - comparing them to Python. We will find them to be very similar, with relatively predictable syntax variations based on the Java that we have learned to this point.

**Conditionals**

Conditional statements in Python and Java are very similar. In Python we have three patterns.

**Simple if**

Let's consider an if statement with no else clause. Here's the Python version.

if condition:

statement1

statement2

...

In Java this same pattern is simply written as:

if (condition) {

statement1

statement2

...

}

Once again, you can see that in Java the curly braces, rather than indentation, define a block. In Java, the parentheses around the condition are required.

**if-else**

Adding an else clause, we have:

if condition:

statement1

statement2

...

else:

statement1

statement2

...

In Java this is written as:

if (condition) {

statement1

statement2

...

} else {

statement1

statement2

...

}

**elif / else if**

In Java we can utilize the same behavior that elif provides in Python, with a slightly different syntax. Here is a simple example in both Python and Java.

grade = int(input('enter a grade'))

if grade < 60:

print 'F'

elif grade < 70:

print 'D'

elif grade < 80:

print 'C'

elif grade < 90:

print 'B'

else:

print 'A'

In Java we would write this as follows:

import java.util.Scanner;

public class ElseIf {

public static void main(String args[]) {

Scanner in = new Scanner(System.in);

System.out.println('enter a grade');

int grade = in.nextInt();

if (grade < 60) {

System.out.println('F');

} else if (grade < 70) {

System.out.println('D');

} else if (grade < 80) {

System.out.println('C');

} else if (grade < 90) {

System.out.println('B');

} else {

System.out.println('A');

}

}

}

**switch**

Java also supports a switch statement that acts something like the Python elifstatement under certain conditions. The switch statement is not used very often, and we generally recommend you avoid using it. It is not as powerful as the else if model because the switch variable can only be compared for equality with a very small class of types.

Here is a quick example of a switch statement:

import java.util.Scanner;

public class DayPrinter {

public static void main(String[] args) {

Scanner in = new Scanner(System.in);

System.out.println("Enter an integer: ");

int dayNum = in.nextInt();

String day;

switch (dayNum) {

case 0:

day = "Sunday";

break;

case 1:

day = "Monday";

break;

case 2:

day = "Tuesday";

break;

case 3:

day = "Wednesday";

break;

case 4:

day = "Thursday";

break;

case 5:

day = "Friday";

break;

case 6:

day = "Saturday";

break;

default:

// in this example, this block runs if none of the above blocks match

day = "Int does not correspond to a day of the week";

}

System.out.println(day);

}

}

In the example above, if the user entered the number *4*, "Thursday" is what would be printed. If the user entered the number *10*, "Int does not correspond to a day of the week" would be printed.

Additionally, if break statements are omitted from the individual cases on accident, a behavior known as [fallthrough](https://en.wikipedia.org/wiki/Switch_statement#Fallthrough) is carried out. Fallthrough can be quite unintuitive, and is only desirable in very specific circumstances. We will discuss break statements in more detail in the loop section below, but for now just know that when used in a switch block they terminate the switch statement they are in, so the flow of control in your program moves to the next statement after the switch block.

Here's a quick example of how fallthrough works:

import java.util.Scanner;

public class DayPrinter {

public static void main(String[] args) {

System.out.println("Enter an integer: ");

Scanner in = new Scanner(System.in);

int dayNum = in.nextInt();

String day;

switch (dayNum) {

case 0:

day = "Sunday";

case 1:

day = "Monday";

case 2:

day = "Tuesday";

case 3:

day = "Wednesday";

case 4:

day = "Thursday";

case 5:

day = "Friday";

case 6:

day = "Saturday";

default:

// in this example, this block runs even if one of the above blocks match

day = "Int does not correspond to a day of the week";

}

System.out.println(day);

}

}

This time, without the break statements in each case, if the user entered *4*, "Int does not correspond to a day of the week" would be printed. This is because after the switch statement matches the case for *4* and assigns the value "Thursday" to the variable day, it proceeds to execute every statement in every case that follows, all the way through the default case. So the String that ends up being printed will reflect the last executed statement in the switch block.

As another variation on this example, if there were a break statement in case 6 after day = "Saturday";, then if the user entered *4*, the execution would fallthrough until it reached that break statement and "Saturday" is what would be printed instead.

**Iteration**

At a conceptual level, loops in Java aren't that different from loops in Python, though the syntax varies significantly in some cases.

**For Loops**

In Python the easiest way to write a definite loop (aka a for loop) is using the for loop in conjunction with the range function. For example:

for i in range(10):

print i

In Java we would write this as:

for (int i = 0; i < 10; i++ ) {

System.out.println(i);

}

You may not be familiar with the expression i++ since it is not found in Python. The ++ is an increment operator that has the same effect as i += 1. In this example, since the ++ comes after i, we call it a postfix increment operator. There is also a -- decrement operator in Java. For more information, see [Increment and Decrement Operators](http://www.javawithus.com/tutorial/increment-and-decrement-operators).

Recall that the range function provides you with a wide variety of options for controlling the value of the loop variable.

range(stop)

range(start,stop)

range(start,stop,step)

The Java for loop is analogous to the last option giving you explicit control over the starting, stopping, and stepping in the three clauses inside the parentheses. You can think of it this way:

for (start clause; stop clause; step clause) {

statement1

statement2

...

}

If you want to start at 100, stop at 0 and count backward by 5 the Python loop would be written as:

for i in range(100,-1,-5):

print i

In Java we would write this as:

for (int i = 100; i >= 0; i -= 5) {

System.out.println(i);

}

In Python the for loop can also iterate over any sequence such as a list, a string, or a tuple. Java also provides a variation of its for loop that provides the same functionality in its so-called *for-each* loop, or *for-in* loop.

In Python we can iterate over a list as follows:

k = [1, 1, 2, 3, 5, 8, 13, 21]

for i in k:

print i

In Java, this would look like:

int k[] = {1, 1, 2, 3, 5, 8, 13, 21};

for(int i : k) {

System.out.println(i);

}

This version translates nicely to iterating through a string, where we can convert the string to an array of characters:

String msg = "Hello World";

for (char c : msg.toCharArray()) {

System.out.println(c);

}

**While Loops**

Both Python and Java support the while loop, or indefinite loop. Recall that in Python the while loop is written as:

while condition:

statement1

statement2

...

In Java we add parentheses and curly braces to get:

while (condition) {

statement1

statement2

...

}

Java adds an additional, if seldom used, variation of the while loop called the do-while loop. The do-while loop is very similar to while except that the condition is evaluated at the end of the loop rather than the beginning. This ensures that a loop *will be executed at least one time*. Some programmers prefer this loop in some situations because it avoids an additional assignment prior to the loop. For example:

do {

statement1

statement2

...

} while (condition);

**Break and Continue Statements**

There are instances where you may want to terminate a loop if a given condition is met. In these instances, the break statement comes in handy. For example, if you want to loop through an array of integers searching for a given integer and you want to quit the loop once that number is found, you can do the following:

public class testBreak {

public static void main(String [] args) {

int[] someInts = {1, 10, 2, 3, 5, 8, 10};

int searchTerm = 10;

for(int oneInt : someInts) {

if (oneInt == searchTerm) {

System.out.println("Found it!");

break;

}

}

}

In the code above, instead of the for loop iterating through all the integers in the array, it will stop after it finds the first matching instance. So once it finds the first *10* in the array, it prints "Found it!" and then terminates the loop. If the break statement weren't there, the loop would continue and when it found the second *10* it would print "Found it!" a second time.

Note that the break statement terminates the innermost loop that it is contained within. So if you have nested loops and use a break statement within the innermost loop, then it will only terminate that loop and not the outer one.

The continue statement is similar to, but importantly different from, the break statement. Like break, it interrupts the normal flow of control of the loop. But unlike break, the continue statement only terminates the *current iteration* of the loop. So the loop will continue to run from the top (as long as the boolean expression that controls the loop is still true) after a continue statement. Here is an example:

public class testContinue {

public static void main(String [] args) {

int[] someInts = {1, 10, 2, 3, 5, 8, 10};

int searchTerm = 10;

for(int oneInt : someInts) {

if (oneInt == searchTerm) {

System.out.println("Found it!");

continue;

}

System.out.println("Not here");

}

}

}

The above program will print "Not here" on every iteration of the for loop *except* where the number has been found. So the output looks like this:

Not here

Found it!

Not here

Not here

Not here

Not here

Found it!

Because of the continue statement, the final print statement in the for loop is skipped. If the continue statement weren't there, the output would look like this instead (notice the extra "Not here" printouts):

Not here

Found it!

Not here

Not here

Not here

Not here

Not here

Found it!

Not here

**References**

* [The if-then and if-then-else Statements (docs.oracle.com)](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/if.html)
* [The switch Statement (docs.oracle.com)](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/switch.html)
* [The for statement (docs.oracle.com)](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/for.html)
* [The while and do-while Statements (docs.oracle.com)](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/while.html)
* [Break and Continue Statements (docs.oracle.com)](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/branch.html)
* [Summary of Control Flow Statements (docs.oracle.com)](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/flowsummary.html)

**Data Structures and Collections**

A **data structure** is a programming construct that allows you to aggregate lots of values into one value. More simply, a data structure lets us hold on to lots of data in a single place. In Python, the data structures we used were lists, dictionaries, and tuples.

Java provides powerful and flexible structures to store data, known as [collections](http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html). We'll introduce only a few here, but they will be sufficient for all of your basic needs while you get going with Java.

**Ordered Data: Lists and Arrays**

We'll explore collections in Java by looking at different versions of the same program. The program will function as a gradebook, allowing a user (a professor or teacher) to enter the class roster for a course, along with each student's grade. It then prints the class roster along with the average grade. In each variation of this program, the grading system could be anything numeric, such as a 0.0-4.0 point scale, or a 0-100 percentage scale.

We'll look at lists first, and as before, we'll compare Python to Java explicitly through example. Python provided one basic *ordered* data structure: the list. Here's our gradebook program in Python using only lists.

The built-in function enumerate is used in the Python example below. In case you aren't familiar with this function, read up about it [here](https://docs.python.org/3/library/functions.html#enumerate).

**Gradebook (Python List Version)**

def main():

students = []

# Use a space to allow for the while check below

newStudent = " "

print("Enter your students (or ENTER to finish):")

# Get student names

while (newStudent != ""):

newStudent = input()

if newStudent != "":

students.append(newStudent)

# Get student grades

grades = [0]\*len(students)

for idx,student in enumerate(students):

newGrade = float(input("Grade for " + student + ": "))

grades[idx] = newGrade

# Print class roster

print("\nClass roster:")

for idx,student in enumerate(students):

print(student + " (" + str(grades[idx]) + ")")

avg = sum(grades) / len(grades)

print("\nAverage grade: " + str(avg))

if \_\_name\_\_ == '\_\_main\_\_':

main()

A test run of this program might yield the following:

Enter your students (or ENTER to finish):

Chris

Jesse

Sally

Grade for Chris: 3.0

Grade for Jesse: 4.0

Grade for Sally: 3.5

Class roster:

Chris (3.0)

Jesse (4.0)

Sally (3.5)

Average grade: 3.5

Once you feel like you understand this program, proceed to the next section, where we study the Java version.

**Gradebook (Java ArrayList Version)**

To write the Java version of this program we will have to introduce several new Java concepts. We will see the Java equivalent of a list, provided by the class ArrayList. We will also review different kinds of for loops used in Java.

package org.launchcode.java.demos.java4python;

import java.util.ArrayList;

import java.util.Scanner;

public class Gradebook {

public static void main(String[] args) {

ArrayList<String> students = new ArrayList<>();

ArrayList<Double> grades = new ArrayList<>();

Scanner in = new Scanner(System.in);

String newStudent;

System.out.println("Enter your students (or ENTER to finish):");

// Get student names

do {

newStudent = in.nextLine();

if (!newStudent.equals("")) {

students.add(newStudent);

}

} while(!newStudent.equals(""));

// Get student grades

for (String student : students) {

System.out.print("Grade for " + student + ": ");

Double grade = in.nextDouble();

grades.add(grade);

}

// Print class roster

System.out.println("\nClass roster:");

double sum = 0.0;

for (int i = 0; i < students.size(); i++) {

System.out.println(students.get(i) + " (" + grades.get(i) + ")");

sum += grades.get(i);

}

double avg = sum / students.size();

System.out.println("Average grade: " + avg);

}

}

Before going any further, I suggest you run the above program in IntelliJ.

Once you've done that, let's look at what is happening in the Java source code.

ArrayList<String> students = new ArrayList<>();

ArrayList<Double> grades = new ArrayList<>();

Scanner in = new Scanner(System.in);

String newStudent;

Here we declare and initialize two objects, students and grades, which appears to be of type ArrayList<String> and ArrayList<Double>, respectively. An ArrayList in Java is very similar to a list in Python, with one important difference. Unlike Python, where lists can contain any type of value, in Java we must let the compiler know what kind of objects our list is going to contain. In the case of students, the ArrayList will contain values of type String (representing the names of the students), so we use the ArrayList<String> syntax to inform the compiler that we intend to fill our list with strings. Similarly, grades will hold exclusively values of type Double and is declared to be of type ArrayList<Double>.

Notice that we declared grades to be of type ArrayList<Double>, using the wrapper class Double rather than the primitive type double. All values stored in Java collections must be objects, so we'll have to use wrapper classes in those situations. This is the one major exception to our rule-of-thumb that primitives are preferred over wrapper types.

We then initialize each list by creating a new, empty list. Note that when we call the ArrayList constructor, as in new ArrayList<>(), we don't need to specify type (it's implicit in the left-hand side of the assignment).

You will sometimes see the ArrayList class written as ArrayList<E>, where E represents a placeholder for the type that a programmer will declare a given list to hold. This is especially true in documentation. You can think of E as representing an arbitrary type.

Classes like ArrayList<E> that take another type or class as a parameter are referred to as **generic classes** or **generic types**.

We then use a do-while loop to collect the names of each of the students in the class.

// Get student names

do {

newStudent = in.nextLine();

if (!newStudent.equals("")) {

students.add(newStudent);

}

} while(!newStudent.equals(""));

Recall that a do-while loop is very similar to a while loop, but it has its check at the end of the loop block. This has the net effect that we'll always run the block at least once. In this example, we prompt the user for a name, which we read in via in.nextLine() when they hit the enter key. To finish entering names, they enter a blank line.

For each student that is entered (that is, each non-empty line), we add the new string to the end of our list with students.add(newStudent). The .add() method is a method provided by the [ArrayList Class](http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html). There are lots of other list methods.

Below the do-while loop are two different for loops that demonstrate the two ways you can loop through a list in Java. Here's the first, which collects the numeric grade for each student:

// Get student grades

for (String student : students) {

System.out.print("Grade for " + student + ": ");

Double grade = in.nextDouble();

grades.add(grade);

}

This for loop syntax is very similar to that of Python, where the analogous loop would begin: for student in students:. As you might expect at this point, we must declare the iterator variable student in Java, which was not explicitly done in Python.

The other for loop on display prints out each student's name and grade:

// Print class roster

System.out.println("\nClass roster:");

double sum = 0.0;

for (int i = 0; i < students.size(); i++) {

System.out.println(students.get(i) + " (" + grades.get(i) + ")");

sum += grades.get(i);

}

In this loop, we use a *loop index*, a different style of for loop. We also introduce the syntax students.size() which utilizes the ArrayList's size() method to return the integer representing the number of items in the list.

The syntax of this for loop may look strange to you, but in fact it is not too different from what happens in Python using range. The syntax for (int i = 0; i < students.size(); i++) is exactly equivalent to the Python for i in range(len(students)). The first statement inside the parenthesis declares and initializes a loop index variable i. The second statement is a Boolean expression that is our exit condition. In other words, we will keep looping as long as this expression evaluates to true. The third statement is used to increment the value of the loop index variable at the end of iteration through the loop. The syntax i++ is Java shorthand for i = i + 1. Java also supports the shorthand i-- to decrement the value of i. Like Python, you can also write i += 2 as shorthand for i = i + 2.

In the final lines of the program, we compute the average grade for all students:

double avg = sum / students.size();

System.out.println("Average grade: " + avg);

**ArrayList Methods and Properties**

Let's gather up a few of the ArrayList methods and properties that we've encountered so far, along a few new ones. While these will be the most common methods and properties that you use with this class, they by no means represent a complete list. Refer to the [official documentation on the ArrayList class](http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html) for such a list, and for more details.

| **Name** | **Description** | **Example** |
| --- | --- | --- |
| size | Represents the number of items in the list, as an int | students.size() |
| add | Adds an item to the list | students.add("Sally") |
| contains | Checks to see if the list contains a given item, returning a boolean | students.contains("Haley") |
| indexOf | Looks for an item in a list, returns the index of the first occurrence of the item if it exists, returns -1 otherwise | students.indexOf("Zach") |
| sort | Sorts a list, using the "default" sort comparison | students.sort() |
| toArray | Returns an array containing the elements of the list | students.toArray() |

**Gradebook (Java Array Version)**

We were introduced to arrays in Java in a previous lesson, so let's spend a moment comparing them to lists. As we noted at that beginning of this section, we are going to use the Java ArrayList type to store simple sets of data. They are easy to use and more closely match the way that Python lists behave.

Why does Java have both arrays and ArrayList? The answer is historical, at least in part. Java is a C-style language, and arrays are the most basic data structure in C. Additionally, using an array over an ArrayList might be preferred in some circumstances, primarily for performance reasons (array operations are generally faster than list operations). Also note that *arrays are of fixed size*. You can not expand or contract an array after it is created, so you must know exactly how many elements it will need to hold when you create it. This fact is reason enough to use lists in most scenarios.

To illustrate array usage, here is a version of the gradebook program that uses arrays instead of lists:

package org.launchcode.java.demos.java4python;

import java.util.ArrayList;

import java.util.Scanner;

public class GradebookArray {

public static void main(String[] args) {

// Allow for at most 30 students

int maxStudents = 30;

String[] students = new String[maxStudents];

double[] grades = new double[maxStudents];

Scanner in = new Scanner(System.in);

String newStudent;

int numStudents = 0;

System.out.println("Enter your students (or ENTER to finish):");

// Get student names

do {

newStudent = in.nextLine();

if (!newStudent.equals("")) {

students[numStudents] = newStudent;

numStudents++;

}

} while(!newStudent.equals(""));

// Get student grades

for (int i = 0; i < numStudents; i++) {

System.out.print("Grade for " + students[i] + ": ");

double grade = in.nextDouble();

grades[i] = grade;

}

// Print class roster

System.out.println("\nClass roster:");

double sum = 0.0;

for (int i = 0; i < numStudents; i++) {

System.out.println(students[i] + " (" + grades[i] + ")");

sum += grades[i];

}

double avg = sum / numStudents;

System.out.println("Average grade: " + avg);

}

}

Note that we have to decide up front how large our arrays students and grades are going to be. Thus, it is advisable to make the arrays potentially larger than they need to be. Like lists, we can index into arrays with integers (students[i] for example). Unlike lists, however, there is no analog of .add(), which adds an item to "the end" of a list. We must always access and assign array elements using an explicit index. This makes for code that can seem littered with array counters (like our friends i and j) and is more difficult to read (not to mention more error-prone).

Like lists, however, we can loop through an array using a for-each loop as long as we don't need to use the index of the current item. If we only wanted to print each student's name, and not their grade, at the end of our program, we could do the following:

for (String student : students)

{

System.out.println(student);

}

We'll use an array from time-to-time, but for the most part you should rely on lists to store collections of values, or ordered data.

**Key/Value Data: HashMaps**

Just as Python provides the dictionary structure to allow us to store data as key/value pairs, Java also provides us a similar mechanism. Java calls these objects hash maps (or maps, more generally), and they are provided by the HashMap class.

Considering the gradebook example, we can improve our program using a map, and storing students' grades along with their names in the same data structure. The names will be the keys, and the grades will be the values.

**Gradebook (Python Dictionary Version)**

Were we to write our gradebook program above using dictionaries, we'd come up with something like this:

def main():

students = {}

newStudent = " "

print("Enter your students (or ENTER to finish):")

# Get student names

while (newStudent != ""):

newStudent = input("Name: ")

if newStudent != "":

newGrade = float(input("Grade: "))

students[newStudent] = newGrade

# Print class roster

print("\nClass roster:")

for student in students:

print(student + " (" + str(students[student]) + ")")

avg = sum(students.values()) / len(students)

print("\nAverage grade: " + str(avg))

if \_\_name\_\_ == '\_\_main\_\_':

main()

**Gradebook (Java HashMap Version)**

Let's now turn to the Java version, using instances of the HashMap class. As with lists, in Java we must specify the types of the objects we'll be storing when we declare a variable or parameter to be a map. This means specifying both key and value data types, which are allowed to be different types for a given map.

package org.launchcode.java.demos.java4python;

import java.util.HashMap;

import java.util.Map;

import java.util.Scanner;

/\*\*

\* Created by LaunchCode

\*/

public class GradebookHashMap {

public static void main(String[] args) {

HashMap<String, Double> students = new HashMap<>();

Scanner in = new Scanner(System.in);

String newStudent;

System.out.println("Enter your students (or ENTER to finish):");

// Get student names and grades

do {

System.out.print("Student: ");

newStudent = in.nextLine();

if (!newStudent.equals("")) {

System.out.print("Grade: ");

Double newGrade = in.nextDouble();

students.put(newStudent, newGrade);

// Read in the newline before looping back

in.nextLine();

}

} while(!newStudent.equals(""));

// Print class roster

System.out.println("\nClass roster:");

double sum = 0.0;

for (Map.Entry<String, Double> student : students.entrySet()) {

System.out.println(student.getKey() + " (" + student.getValue() + ")");

sum += student.getValue();

}

double avg = sum / students.size();

System.out.println("Average grade: " + avg);

}

}

We can add a new item with a .put() method, specifying both key and value: students.put(newStudent, newGrade).

And while we don't do so in this example, we may also access HashMap elements using the get method. If we had a key/value pair of "jesse"/4.0 in the students map, we could access the grade with Double jesseGrade = students.get("jesse"). As with Python, variables may be used to access elements:

String name = "jesse";

Double jesseGrade = students.get(name);

Looping through a map is slightly more complex than it is for lists. Let's look at the for-each loop from this example:

for (Map.Entry<String, Double> student : students.entrySet()) {

System.out.println(student.getKey() + " (" + student.getValue() + ")");

sum += student.getValue();

}

The iterator variable, student, is of type Map.Entry<String, Double>. The class Map.Entry is specifically constructed to be used in this fashion, to represent key/value pairs within HashMaps. Each Map.Entry object has a getKey method and a getValue method, which represent (surprisingly enough!), the key and value of the map item.

If you only need to access the key of each item in a map, you can construct a simpler loop:

for (String student : students.keySet())

{

System.out.println(student);

}

A similar structure applies if you only need the values, using students.values():

for (double grade : students.values())

{

System.out.println(grade);

}

**HashMap Methods**

Let's collect some HashMap properties and methods. As we said about lists, this is by no means a comprehensive list. For full details on all properties and methods available, see the [official documentation on the HashMap class](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html).

| **Name** | **Description** | **Example** |
| --- | --- | --- |
| size | Returns the number of items in the map, as an int. | students.size() |
| keySet | Returns a collection containing all keys in the map. This collection may be used in a for-each loop just as lists are, but the map *may not be modified* within such a loop. | students.keySet() |
| values | Returns a collection containing all values in the map. This collection may be used in a for-each loop just as lists are. | students.values() |
| put | Add a key/value pair to a map. | students.put("Mark", 3.5) |
| containsKey | Returns a boolean indicating whether or not the map contains a given key. | students.containsKey("Chris") |
| containsValue | Returns a boolean indicating whether or not the map contains a given value. | students.containsValue(4.0) |

We have only brushed the surface of how lists and maps work, and we leave it to you to refer to the official documentation linked below for more details. You'll certainly be using lists and maps in more ways than those covered in this lesson, but with the knowledge you have now, you should be able to use Java collections and learn new uses as you go.

**References**

* [Java Collections (docs.oracle.com)](http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html)
* [ArrayList Class (docs.oracle.com)](http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html)
* [HashMap Class (docs.oracle.com)](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)
* [Arrays Tutorial (docs.oracle.com)](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html)