2 Java Basics (For Python Users)

This chapter discusses basic Java features, enough so that we can begin using the language to talk about and implement data structures. It was written assuming the readers are familiar with Python although readers proficient in other programming languages should have no trouble picking up Java from the exposition in this chapter.

We will begin by writing an obligatory "Hello World" program. After that, Java features will be introduced in turn. This includes types in Java, control-flow constructs, declaring functions, and fixed-size arrays. At the end of the chapter, readers are expected to be comfortable writing basic Java.

Why should we bother learning Java? For readers coming from Python, this is an often-asked question. Java is extremely popular in large-scale software systems. Compared to Python, Java tends to be faster and the fact that Java is a statically-typed language (more on this later) makes it easier to catch and fix errors before the code is run. It is also an excuse to learn object-oriented programming along the way.

2.1 Our First Java Program: Hello, World

A time honored tradition in Computer Science is to write a program called hello world. Back in Python, this is just a one-liner:

```
print("Hello, World!")
```

What does it take to accomplish the same thing in Java? The nature of Java requires a bit more code and more structure:

```
public class Hello { // Save this in Hello.java
    public static void main(String args[]) {
        System.out.println("Hello, World!");
    }
}
```

The identifier (name) after `public class` indicates that this is a public class named `Hello`. Java expects every public class to be kept in a `.java` file with the same name. Therefore, this program will have to saved in `Hello.java`

Compile/Run. But how do we run it? Running a Java program involves more steps than running a Python program. The first step is to compile it.

This is the first big difference between Java and Python: Python is (mainly) an interpreted language. We instruct a Python interpreter to run our Python programs—and we obtain the results. In a normal workflow, Java makes running a program a 2-step process:

- First, translate it into a representation that is close to what the machine understands. This is called compilation, accomplished on the command-line like so:

```
javac Hello.java
```

Notice that the name of the class is the same as the name of the file. This will produce `Hello.class`, which is the binary version of the Java source file we wrote.

Technically, this is not true after Java 9, but still, a normal workflow involves proper compilation. In Java 9 and above, we can readily call `java Hello.java`.

- Then, we can run it. Java does not quite compile everything down to the level that the machine can understand. It compiles the code down to a representation known as the *bytecode*, a Java's internal representation that has to be run using Java. To actually run our program, we will call

```
java Hello
```

What good is this extra step? In particular, what does compiling do for us? There are a couple of important benefits we get from compiling:

- **Early detection of errors.** For many types of errors (e.g., syntax errors or simple logic errors such as incorrect types), the compilation step can detect it before the program is run. In this way, we do not have to waste time running it to spot a silly mistake.
- **Faster Program Execution.** Because the program has been translated into bytecode (a format that is more convenient for Java to work with), Java can run this program faster. Intuitively, after a round of compilation, Java does not have to think as much as a normal interpreter; it can just run the commands.

Main: The Entry Point Into Our Program

Now that we have run our hello world program, we will go back and look at it carefully to see what we can learn about this Java program. This simple example illustrates a few very important points:

- Every Java program must live in a class; all code is inside a class.
- The entry point to a Java program is a function called

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

In the above code, we defined a class `Hello`. It has one function (more properly known as a *static method*). This is not just any function. It is a function called `main` with a specific signature that makes it the *main function*—the entry point into our program.

Everything on that line is important and is there for a reason. We will learn about these features later. For now, notice that a similar-looking functions such as the following are **all** different functions than the main function that is the entry point:

```
public void main(String[] args) // missing static
static void main(String args)   // missing public, missing []
```

Scoping Rule. We will now move on to the body of the function `main`. We know that this is the body of the function because it is enclosed by a pair of *curly braces*. We may remember that in Python, indentation marks the scope of a block of code, be that the code inside a for-loop or a function.

Java uses different notation: a scope begins with an open curly brace `{` and ends with a close curly brace `}`. Therefore, we can spot the body of the `main` function above by locating where we see an open brace and where we see a closing brace.

Dot Notation and Semicolon. As we continue to examine the body of the function, we see exactly one statement: `System.out.println(...)`. This should be familiar to you: both Python and Java use the dot notation for finding names. In particular, the dot notation in this example means:

- We start with `System`, which is a class. Within this class, there is an object (think of it as a box of treasures) called `out`. This is the object that corresponds to activities about standard out—the default console output.
- Inside it, we call the method/function `println`. This is similar to the command `print` in Python except Java's version is much more explicit.
- The function/method `println` prints the given string and starts a new line at the end.
- As a rule of thumb: wherever you want to use Python's `print`, use `System.out.println` or a more fancy cousin `System.out.printf`.

If we look closely, this line has not ended yet. There is one more character at the end—the semicolon `;`. In Java, the `;` signifies the end of a statement. Back in Python, each statement lives on its own line. In Java, this may not be the case. It is acceptable (and sometimes idiomatic) to put several statements on the same line.

Also, even though it may not look it, the following statements are valid. For simple code like this, no one in the right mind would write it as such, but as we see more Java programs, we will see why such freedom in expressing a statement can be a big win in more complex code:

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

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

Comments. There are two forms:

- `//` — is a single line comment. Use it like so:

```
1 // a comment line by itself
2 System.out.println("Hello"); // after a statement
```

- The other form is for a longer piece of text, like in this example:

```
1 /* This line and even some more text is comment out.
2    Also this line.
```

```
   also this line.
*/
```

Java treats everything between the opening `/*` and the closing `*/` markers as a comment.

2.2 Type Declaration

We begin by discussing the first important technical consideration when picking up Java given some familiarity with another language. In Python, we would simply write `x = 5` to set the value 5 to `x`. In Java, we will have to be slightly more explicit with letting Java know the type. To do this, we will write:

```
int x = 5;
```

This statement says declare a variable `x`. This is going to have type `int`. Moreover, set it to `5`.

It is also possible to separate the statement that declares our variable from the statement that assigns a value to it, as this example illustrates:

```
int a;  // declare a variable a without setting it to a value

a = 28; // assign 28 to a
```

Other than the `int`s, there are several other common types:

```
byte    short    int    long  // integers
char                          // character
boolean                       // true/false Boolean
float   double                // floating-point numbers
```

Several important remarks are in order: There are many types for whole numbers (integers): `byte`, `short`, `int`, `long`. They differ in two important ways: (1) they are stored using different numbers of bits; and (2) they can maintain different ranges of numbers.

Remember? In contrast to whole numbers, floating-point numbers such as 1.0, 3.14, and 2.718 can have a decimal point. Integer types store whole-number values such as $0, 1, 2, -5$.

For example, a `byte` occupies only 1 byte (8 bits) of memory and can store a number between `-128` and `127` (inclusive). An `int` occupies 4 bytes (32 bits) of memory and can store a number between -2^{31} to $2^{31} - 1$ (inclusive).

In addition, there are two pre-made types for floating-point numbers: `float` and `double`. They differ in how much precision they can carry, with `double` using 2x the number of bits used by `float`.

Readers are encouraged to check out the primitive data types tutorial from the official Java documentation for detailed information.

2.3 Static Typing

Unlike Python, Java is statically typed. What does this mean exactly?

- All variables, parameters, and methods must have a declared type. Moreover, that type can never change.
- Expressions also have a type. For example, `max(5, 10) + 3` has type `int`. The compiler checks that all the types in our program are compatible before the program ever runs! This means, for instance, the following code will not compile.

  ```
  int x = max(5, 10) + "3"
  ```

  Java knows not to let this through because `max(5, 10)`, as we expect, gives us a whole number (an `int`), so adding that to a string `"3"` will not really work. On top of that, trying to store the result into a variable of type `int` is just not going to work.
- Static typing is unlike a language like Python, where type checking is performed during execution. If this were Python, we wouldn't find this out until we run the code and see the dreaded `TypeError` exception.

Example: Simple Temperature Conversion

To practice what have seen so far, we will write a program that uses some of these features. We will write a program to convert a floating-point number (`double`) representing the temperature in Celsius (C) into a floating-point number (`double`) representing the temperature in Fahrenheit (F). Finally, we will print out the answer nicely.

There are 3 steps in this logic:

1. Declare a variable to store the temperature in C.
2. Declare a variable to store the temperature in F and perform the calculation. It helps to remember that $F = 32 + C \cdot \frac{9}{5}$.
3. Print out the answer.

In code:

```
double tC = 32.751;            // Step 1
double tF = 32.0 + tC*9.0/5.0  // Step 2
System.out.println(tF);        // Step 3
```

Notice that this is not yet a proper Java program because all these have to be put inside a function (method) and a class. Below is an example of a complete program, where we put this code inside `main` inside a `BasicTempConvert` class.

```
1 // save this to "BasicTempConvert.java"
2 public class BasicTempConvert {
3     public static void main(String args[]) {
4         double tempC = 32.751;
5         double tempF = 32.0 + tempC*9.0/5.0;
6
7         System.out.println(tempF);
8     }
9 }
```

Python Users Be Warned: In Java, the expression `10/3` is not 3.33333 . . ., like in Python 3. It is an `int` of value `3`.

2.4 Functions (technically, static methods)

We have written functions (or really static methods) in Java before. Remember the `main` function? What we wrote was:

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

Let us break this down a bit:

The Public Keyword. The keyword `public` makes this visible outside of this class. Without it, our function will not always be visible. In this particular case, without it, `main` will not be visible as an entry point to our program. Later in the book, we will take a closer look at "visibility" modifiers.

Static vs. Standard Methods. Because all code belongs to a class, Java does not really have a notion of free-floating functions. Function-like code unit that lives inside a class is known as a *method*. There are two kinds of methods:

- **Static methods**, or class methods, are, to the extent of our understanding right now, similar to our beloved functions; and
- **(Standard) methods**, which we will study in greater detail later.

Until then, the only kind of methods we are writing is `static`. Static methods need the keyword `static`.

The Void Keyword. The keyword `void` indicates the return type of the function. `void` says the function doesn't return anything useful (similar to returning `None` in Python). More generally, a function can return an `int`, a `double`, and a `long`, etc.

Function Name and Arguments. In this example, `main` is the name of the function. In general, Java prefers Camel-case naming*, for example, writing `helloFunWorld` as opposed to `hello_fun_world`.

The stuff in parens—i.e., `(String[] args)`—are parameters (arguments). The parameters are comma separated and each has the form: `typeName varName`. Here are some examples:

```
// a public add "function" that takes 3 int parameters
// and returns an int
public static int add(int a, int b, int c)

// a public fooBar "function" that takes an int and a double
// as parameters and returns a byte
public static byte fooBar(int secretCode, double magicRatio)
```

*See, for example, `https://en.wikipedia.org/wiki/Camel_case`.

```
// a public magic "function" that takes no parameters and
// returns a String
public static String magic()
```

Return Values. Inside a function, the command `return` behaves in the same way as in Python, so if we want the number 42 returned, we will write `return 42;` We will need that semicolon.

Let us now take the temperature conversion example and rewrite it as a static method:

```
1 public static double convertCtoF(double C) {
2     return 32.0 + C*9.0/5.0;
3 }
```

2.5 Conditionals

The `if` statement in Java checks the condition inside the parentheses, and if the result is true (if the condition is met), it executes the next statement below. The syntax is

```
if (<boolean expression>) {
    ...
}
```

Notice that the parentheses around the expressions are *always* needed. Here is a brief example:

```
if (x > 10) {
    // do this if x > 10 is true
    // ...
    // ...
}
```

For an example, play with the code below to get a better understanding:

```
1  public static void main(String[] args) {
2      int x = 5;
3
4      if (x < 10)
5          x = x + 10;
6
7      if (x < 10)
8          x = x + 10;
9
10     System.out.println(x);
11 }
```

Curly Braces? We can have multiple statements in response to a single condition. To do this, wrap the statements in curly braces, for example:

```
int x = 5;

if (x < 10) {
    System.out.println("I shall increment x by 10.");
    x = x + 10;
}

if (x < 10) {
    System.out.println("I shall increment x by 10.");
    x = x + 10;
}

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

We may wish to run it to see the effects.

Python Users Be Warned: Curly braces are very important in Java! Unlike in Python, statements in Java are grouped by braces, and **not** by indentation. For an example of how this can go terribly wrong, the code below is supposed to print out the absolute value of x . That is, if x is positive, print x and if x is negative, print -x. Can you figure out what is wrong?

```
public class PrintAbsoluteValue {
    public static void main(String[] args) {
        int x = -5;

        if (x < 0)
            System.out.println("I should negate X");
            x = -x;

        System.out.println(x);
    }
}
```

A Holy War. Which of the following do you like more?

```
if (x > 5) {
    x = x + 5;
}
```

```
if (x > 5)
{
   x = x + 5;
}
```

Both are acceptable in general. Which of these two styles is "correct" is a bit of a holy war but generally boils down to the house style. In this book, we generally prefer the left style more.

Else

The `else` keyword allows us to specify behavior that should occur if the condition is not met:

```
int x = 9;
if (x - 3 > 8) {
    System.out.println("x - 3 is greater than 8");
} else {
    System.out.println("x - 3 is not greater than 8");
}
```

We can also chain `else` statements, for example:

```
int catSize = 20;

if (catSize >= 50) {
    System.out.println("mroaeww!");
} else if (catSize >= 10) {
    System.out.println("meeoow!");
} else {
    System.out.println("mew!");
}
```

Notice that Java does not have an `elif` keyword. The closest equivalent is to write `else if`.

2.6 Loops and Fixed Arrays

While Loops

Similar to Python, Java's `while` keyword lets us repeat a block of code as long as some condition is true. Below is a typical while-loop program:

```
int bottles = 5;
while (bottles > 0) {
    System.out.println(bottles + " bottles.");
    bottles = bottles - 1;
}
System.out.println("Done.");
```

Program's Output: The while loop repeats itself 5 times.

```
5 bottles.
4 bottles.
3 bottles.
2 bottles.
1 bottles.
Done.
```

This means the block of code that is the body of the while loop will be run as long as the condition guarding the while loop is true.

Fixed-Size Arrays

Fixed-size arrays in Java are like lists in Python except that they have a fixed size. We indicate the size of an array when we create it—and we cannot change it afterward. Below is the same logic implemented in Python and Java (both Python and Java start the index at 0).

```
# Python
a = [3, 1, 4]
print(a[2])  # 4
```

```
// Java
int[] a=new int[]{3, 1, 4};
System.out.println(a[2]);
```

If we wish to know the length of an array, use `.length`, for example:

```
System.out.println(a.length);
```

For Loops

In Python, we are quite used to the idea of writing for-each loops. For example, the Python code below adds up numbers in the list and while doing so, it prints out each number

```
numbers = [3, 1, 4, 2, 8]
total = 0
for num in numbers:
    total += num
    print('---', num)
print(total)
```

To see how this works in action, step through the execution of the code in a code visualizer, for example, `https://pythontutor.com/java.html`. If you're using an IDE such as IntelliJ or Eclipse, the Debug mode can be used to step through each line of code and observe the effects.

The same logic in Java looks as follows:

```
int[] numbers = new int[]{3, 1, 4, 2, 8};
int total = 0;
for (int num: numbers) {
    total += num;
    System.out.println("--- " + num);
}
System.out.println(total);
```

Break and Continue. Occasionally, we will find it useful to use the `break` or `continue` keywords. Both `break` and `continue` work in the same way in Python and Java. Remember that the `continue` statement skips the rest of the current iteration of the loop, effectively jumping straight to the top of the loop. On the other hand, the `break` keyword terminates the innermost loop when it is called.

Just like in Python, `break` and `continue` work with both the `while` and `for` loops.

Full-Blown C-Style For

A common loop pattern is to have:

- an *init* clause (e.g., `int index = 0`) at the top
- a *condition* that has to be true for the loop to run (e.g., `index < numbers.length`); and
- an *increment/update* clause at the bottom of the loop to update the running variable (e.g., `index++`)

This pattern is so common that many languages have dedicated a form of `for` loop for it. In Java, the `for` keyword supports this pattern using the syntax below:

```
for (init; condition; increment) {
    // loop code
}
```

Notice that the clauses are semicolon-separated. For a concrete example, we will rewrite the above while-loop code using this `for` syntax:

```
int[] numbers = new int[]{3, 1, 4, 2, 8};
for (int index=0;index<numbers.length;index++) {
    System.out.println("--- " + numbers[index]);
}
```

Some Observations. Some observations regarding loops are in order:

- The for loop is often easier to read (i.e., cleaner) because it puts all the loop-related statements at the top of the loop.
- There is one difference between for loops and while loops: if we declare a variable in the initializer, it only exists inside the for loop.

More Arrays

Arrays that we saw earlier are known as *fixed-size arrays* or primitive arrays. This is the most basic form of arrays available in Java. In more detail, it is similar to Python's list except that

- it has a fixed size;
- it lacks most of the convenience functions; and
- it can only store elements of the same type. (One cannot keep `int`s and `String`s in the same array. At least, this is advised against.)

Demystifying Types. When we write `int a = 5`, we are declaring a variable a to have type `int` and initialize it to have the value `5`. The `int` in front of the variable name indicates the type.

Putting a pair of open-close brackets `[]` next to `int`—that is, writing `int[]`—gives rise to a new type. This is the type "an array of `int`s." Hence, writing `int[] numbers` is declaring a variable `numbers` of type `int[]`, which is an array of `int`s.

Printing. As it turns out, the following code does not do what we think it does:

```
int[] numbers = new int[]{1, 3, 2, 4, 5};
System.out.println(numbers);
```

On this machine, the following is shown as a result of running the code above:

```
[I@42e26948
```

That is gibberish. Technically, it displays the location where this array is kept, but more on this when we discuss references.

To really display the contents of the array, we need some extra help: Use `Arrays.toString(anArray)` to obtain a `String` representation of the arrray, which is then human readable. Let us try that:

```
int[] numbers = {1, 3, 2, 4, 5};
System.out.println(Arrays.toString(numbers));
```

But wait! where is `Arrays.toString`? It is part of a package that is not imported by default. To use it, we will need to bring it into our view, by putting the following `import` statement at the top of the file:

```
import java.util.Arrays;
```

This imports the `Arrays` class, which is part of `java.util`. When we would like to import multiple things at once, we could write

```
import java.util.*;
```

which says import everything from inside `java.util`.

Creating Empty Arrays. How do we create an array of 10 million `long`s? Typing `long a[] = {0L, 0L, ...}` is probably not going to end well. Instead, to create the array, use the `new` operator:

```
long a[] = new long[10000000];

// OR we can write them separately
long a[];

a = new long[10000000];
```

While the former form defines a variable and sets it equal to a brand new array of the specified size, the latter form defines a variable a without assigning to it—and separately, we create a new array and set it to a.

Do you prefer `int[] m={1, 2};` or `int[] m=new int[]{1, 2};`? They mean the same thing when it comes to initializing an array variable with a constant array.

This suggests that a variable can be assigned and reassigned. In fact, the following code is standard:

```
int[] numbers = {3, 2, 5};
System.out.println(Arrays.toString(numbers));
numbers = new int[4];
System.out.println(Arrays.toString(numbers));
```

It should be said that the `new` operator can take a number from a variable. Thus, it is legit to write the following code where the array size is given as a variable:

```
int n = magicLengthFormula(...);
int[] a = new int[n];
```

Note that when we create a new array using `new`, the elements are initialized to a default value (for integer types, that is 0). We can access and update the contents of an array using the same syntax as in Python. For example, try running this code in the visualizer:

```
int[] a = {7, 1, 1, 9, 8, 2};
int b = a[0] + 5;
a[4] = 25;
a[2]++;
a[3] += 11;
```

Copying an Array. How can we copy an array? As we know from Python, writing `int[] b = a` is not going to get us a new array with the same contents as the array a's contents—it would simply point b to the same array as a. However, Java provides a convenient function for us to copy an array:

```
int[] b = Arrays.copyOf(a, a.length);
```

Alternatively, we can copy it manually by making a new array of the same length and looping through to copy element by element. We will leave this as an exercise.

Quick Primer on Strings

We cannot talk about arrays without talking about strings. In Python, they are very similar. In Java, they are kind of similar but the syntax for working with them turns out to be different.

For now, we will take a look at three features of the string:

- Determine the length of a string;
- Fetch the `i`-th character of a string; and
- Select a substring of a string.

Like in Python, Java's strings are immutable. The following example demonstrates both of these features:

```
String st = "Hello, World!";

// use .charAt(i) to get the i-th element.
// use .length() to determine the length.
for (int i=0;i<st.length();i++) {
    System.out.println("Index " + i + " has " + st.charAt(i));
}
// use .substring to make a substring
String shortStr = st.substring(2, 5);
System.out.println("shortStr: " + shortStr);
```

We remark that the method `.charAt(index)` returns a `char`, not a `String`. The `char` data type works much like an integer type. For more detail, there are excellent references online.

2D Arrays

Java also supports nested arrays. We will start with the most fundamental question:

> What is the type of an array of arrays of `int`s? For example, an array to store `[[1,2,3], [4,5,6]]`.

We know that `int[]` is an array of `int`s, so it shouldn't surprise us to learn that `int[][]` is an array of arrays of `int`s. (The second `[]` says it's an array of `int[]`).

But how do we create one?

Example: Full $n \times n$ Grid

Example. For `n = 3`, the grid will look like this:

```
0, 1, 2
1, 2, 3
2, 3, 4
```

In this example, we are writing a program that will make an `n`-by-`n` grid, filling the `i`-th row at the `j`-th column with a value of `i + j`.

The following code implements the familiar logic—do pay close attention to the line that makes the array, however: `int[][] grid = new int[n][n]`. In one fell swoop, Java lets us create an `n`-by-`n` array of `int`s.

```
public static int[][] makeGrid(int n) {
    int[][] grid = new int[n][n];
    for (int i=0;i<n;++i) {
        for (int j=0;j<n;++j) {
           grid[i][j] = i+j;
        }
    }
    return grid;
}
```

Example: Triangular Array

Example. For `n = 4`, we have:

```
0
0, 1
0, 1, 2
0, 1, 2, 3
```

In certain scenarios, we wish to have rows of uneven lengths. In this example, we will write a function that returns a triangular array with `n` rows. A new trick up the sleeve is to create each row individually, like so:

```
public static int[][] makeTri(int n) {
    int[][] tri = new int[n][]; // make n rows of int[]

    for (int i=0;i<n;++i) {
        // make the i-th row, which has length i + 1
        tri[i] = new int[i+1];
        for (int j=0;j<=i;++j) {
                  tri[i][j] = j;
        }
    }
    return tri;
}
```

Initializing Nested Lists

When generating test cases, we often rely on being able to easily input nested lists. In Python, we are used to writing, for example,

```
testCase = [
  [0],
  [2, 3],
  [],
  [4, 9, 5, 2]
]
```

As it turns out, we can do almost the same in Java using the following pattern:

```
int[][] testCase = new int[][]{
  new int[] {0},
  new int[] {2, 3},
  new int[] {},
  new int[] {4, 9, 5, 2}
};
```

Or even more concisely:

```
int[][] testCase = new int[][]{
  {0},
  {2, 3},
  {},
  {4, 9, 5, 2}
};
```

We remark that the `new int[][]` at the top can't be removed, though.

2.7 Java Documentation

One nice thing about Java is that it comes with an extensive library of classes and methods. But before we can use them, we need to know what they do. This usually means reading the documentation, which can sometimes be daunting to begin.

As an example, we will write a program that computes $n!$, except that n can be as large as a couple of hundreds. The logic is simple, as the following Python code shows:

```
n = 150
n_factorial = 1
for i in range(2, n+1):
    n_factorial *= i
print(n_factorial)
```

How do we write this in Java? The trouble is, we know the result will exceed the range an `int` can manage. In fact, it will also exceed the capacity of a `long`. Fortunately, Java has support for even larger numbers: `BigInteger`[†].

We need an ability to multiply "big numbers" and presumably to convert a normal `int` into a "big number." Browsing the documentation, we can find what we need:

- We can turn a normal number (e.g., a `long`) into a `BigInteger` by calling `.valueOf`.
- We know we can multiply two `BigIntegers` by invoking `a.multiply(b)`.

This suffices for us to write the following program:

```
1 import java.math.BigInteger;
2
```

[†]The documentation for `BigInteger` for Java 11 can be found at `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/math/BigInteger.html`.

```java
public class NFactorial {
   public static void main(String[] args) {
      int n = 150;
      BigInteger nFac = BigInteger.valueOf(1);
      for (int i=2;i<=n;i++) {
           nFac = nFac.multiply(BigInteger.valueOf(i));
      }
      System.out.println(nFac);
   }
}
```

Exercises

Exercise 2.1. Using a for-loop in the "for-each" form, write a method

```
public static int maxFor(int[] numbers)
```

that returns the maximum number in the input array `numbers`.

Exercise 2.2. Using a while-loop, write a method

```
public static int maxWhile(int[] numbers)
```

that returns the maximum number in the input array `numbers`.

Exercise 2.3. Using a full-blown C-style for-loop, write a method

```
public static int maxFullFor(int[] numbers)
```

that returns the maximum number in the input array `numbers`.

Exercise 2.4. Write a public class `Diamond`. Inside it, implement a function

```
public static void printDiamond(int k)
```

that takes a nonnegative integer `k` denoting the size of the diamond and prints to the screen (using `System.out`) a diamond of size `k`.

A diamond of size k contains a total of $2k-1$ lines, where every line contains a total of $2k+1$ characters (each character is either a `#` or a `*`). A few examples are given in Figure 2.1 for you to discover the pattern.

```
  ##*##          ###*###          ####*####
  #***#          ##***##          ###***###
  ##*##          #*****#          ##*****##
                 ##***##          #*******#
                 ###*###          ##*****##
                                  ###***###
                                  ####*####
```

Figure 2.1: Examples of diamonds of sizes $k = 2, 3, 4$, respectively.

Exercise 2.5. Write a static method `numTrailingZeros(int n)` that returns the number of zeros we have at the end of $n!$. For example, `20!` is 2432902008176640000, which has 4 trailing zeros.

Exercise 2.6. Write a static method `windowPosSum(int[] a, int n)` that replaces each element `a[i]` with the sum of `a[i]` through `a[i + n]`, but only if `a[i]` is positive. If there are not enough values because we reach the end of the array, we sum only as many values as we have.

For example, suppose we call `windowPosSum` with the array `a = {1, 2, -3, 4, 5, 4}`, and `n = 3`. In this case, we would:

- Replace `a[0]` with `a[0] + a[1] + a[2] + a[3]`.
- Replace `a[1]` with `a[1] + a[2] + a[3] + a[4]`.
- Not do anything to `a[2]` because it is negative.
- Replace `a[3]` with `a[3] + a[4] + a[5]`. (We cannot go further than the end of the array.)
- Replace `a[4]` with `a[4] + a[5]`.
- Not change the value of `a[5]` because there are no values after `a[5]`. So the sum is `a[5]` itself.

Thus, the result after calling `windowPosSum` would be `{4, 8, -3, 13, 9, 4}`.

As another example, if we called `windowPosSum` with the array `a = {1, -1, -1, 10, 5, -1}`, and `n = 2`, we would have `{-1, -1, -1, 14, 4, -1}`.

Exercise 2.7. You surely have encountered Roman numerals: I, II, III, XXII, MCMXLVI, etc. Roman numerals are represented by repeating and combing the following seven characters: `I` $= 1$, `V` $= 5$, `X` $= 10$, `L` $= 50$, `C` $= 100$, `D` $= 500$, `M` $= 1000$. This is a good time to refresh your memory of how it works by browsing the Internet.

You will write a public class `Roman`. Inside it, implement a function

```
public static int romanToInt(String romanNum)
```

that takes a string that is a number represented using Roman numerals and returns the number, as an integer, equals in value to the input. For example:

- `romanToInt("I")==1`
- `romanToInt("V")==5`
- `romanToInt("VII")==7`
- `romanToInt("MCMLIV")==1954`
- `romanToInt("MCMXC")==1990`

Exercise 2.8. In this exercise, we will play with the beloved Fibonacci sequence. The first two numbers in the sequence are 1 and 1. Each subsequent term is the sum of the previous two. Mathematically, we have that $F_1 = 1$, $F_2 = 1$, and for $n \geqslant 3$,

$$F_n = F_{n-1} + F_{n-2}.$$

Therefore, the initial 12 numbers of the sequence are: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144

As can be seen, the 12-th Fibonacci number, F_{12}, is 144, which has 3 digits.

Your Task: You will write a public class `Fib`, inside which you will implement a function

```
public static int firstNDigit(int n)
```

that takes an integer `n` as input and returns the smallest index `k` such that F_k has at least `n` digits. Before it becomes too large and too slow, we are only interested in $k \leqslant 40,000$. For example:

- `firstNDigit(1)==1`
- `firstNDigit(2)==7`
- `firstNDigit(3)==12`

HINT: Consider using`BigInteger` (or `BigDecimal`) as F_{47} already exceeds the capacity of an `int` and F_{100} is easily beyond the capacity of a `long`.

Exercise 2.9. This exercise involves writing a utility function for fixed arrays. You will make a public class called `Subsel`, inside which you will implement two functions

```
public static int[] takeEvery(int[] a, int stride, int beginWith)
```

and

```
public static int[] takeEvery(int[] a, int stride)
```

with the following specifications:

- The call `takeEvery(a, s, b)` returns an `int` array containing the elements

  `a[b], a[b+s], a[b+2*s], a[b + 3*s], ....`

  until the index in this progression falls out of `a`.
- The call `takeEvery(a, s)` has the same effect as calling `takeEvery(a, s, 0)`.
- The output may be empty; the `stride` parameter could be positive or negative—but never 0.

This means, for example (abusing notation):

- `takeEvery([1, 2, 3, 4], 2) == [1, 3]`
- `takeEvery([2, 7, 1, 8, 4, 5], 3, 2) == [1, 5]`
- `takeEvery([3, 1, 4, 5, 9, 2, 6, 5], -1, 5) == [2, 9, 5, 4, 1, 3]`

Exercise 2.10. Meow is playing a game with her friend. She gives her friend a string `s` and asks her whether another string `t` "hides" inside `s`. Let's be a bit more precise about hiding. Say you have two strings `s` and `t`. The string `t` hides inside `s` if all the letters of `t` appear in `s` in the order that they originally appear in `t`. There may be additional letters between the letters of `s`, interleaving with them.

Under this definition, every string hides inside itself: `"cat"` hides inside `"cat"`. For a more interesting example, the string `"cat"` hides inside the string `"afciurfdasctxz"` as the diagram below highlight the letters of `"cat"`:

```
afciurfdasctxz.
```

Importantly, these letters must appear in the order they originally appear. Hence, this means that `"cat"` does not hide inside the string `"xaytpc"`. In this case, although the letters `c`, `a`, and `t` individually appear, they don't appear in the correct order.

As another example, the string `"moo"` does not hide inside `"mow"`. This is because although we can find an `m` and an `o`, the second `o` doesn't appear in `"mow"`.

In this problem, you will help Meow's friend by implementing a function

```
public static boolean isHidden(String s, String t)
```

that determines whether `t` hides inside `s`. The function returns a Boolean value: `true` if `t` hides inside `s` and `false` otherwise.

Here are some more examples:

```
isHidden("welcometothehotelcalifornia","melon") == true;
isHidden("welcometothehotelcalifornia","space") == false;
isHidden("TQ89MnQU3IC7t6","MUIC") == true;
isHidden("VhHTdipc07","htc") == false;
isHidden("VhHTdipc07","hTc") == true;
```

Chapter Notes

This chapter was inspired by Bradly Miller's open-source book [Mil08] to help Pythonistas more smoothly transition from Python to Java. The material discussed here is elementary and is compatible with recent versions of Java (Java 7 and later). For more in-depth discussion, readers should check out the excellent *Core Java* series, of which *Core Java Volume I–Fundamentals* [Hor16] is an excellent starting point for a deep dive.