Introduction to Java in [49] Pages

Ryan Dougherty
## Contents

1 Computers ................................. 1  
1.1 Motivation ................................ 1  
1.2 Introduction to Java/Programming Languages ................................... 1  
1.3 Java Program Structure ........................ 1  
1.4 Running/Compiling Java Programs ................................... 2  
1.5 Compilation Errors ................................ 3  
1.6 Notes ...................................... 3  
1.7 Written Exercises ............................. 4  
1.8 Programming Exercises .......................... 4  

2 Data Types .................................. 5  
2.1 Types ...................................... 5  
2.2 Variable Declaration ......................... 5  
2.3 Primitive Operators ........................... 6  
2.4 Boolean .................................... 8  
2.5 String ..................................... 9  
2.6 Scanner .................................... 10  
2.7 Math ....................................... 11  
2.8 Written Exercises ............................. 11  
2.9 Programming Exercises .......................... 13  

3 Decisions ................................... 15  
3.1 Motivation .................................. 15  
3.2 If Statements ................................. 15  
3.3 Switch Statements ............................ 16  
3.4 Written Exercises ............................. 18  
3.5 Programming Exercises .......................... 20  

4 Loops ........................................ 21  
4.1 Motivation .................................. 21  
4.2 Introduction to Loops ......................... 21  
4.3 While loops .................................. 21  
4.4 Do-While loops .............................. 22  
4.5 For loops ................................... 23  
4.6 Loop Example ................................ 24  
4.7 Written Exercises ............................. 25  
4.8 Programming Exercises .......................... 26
5 Introduction to Classes 27
  5.1 Class Structure .......................................... 27
    5.1.1 Visibility Modifiers .................................. 27
    5.1.2 Instance Variables .................................... 27
  5.2 Methods .................................................. 31
    5.2.1 Structure .............................................. 31
    5.2.2 Methods Returning Values .............................. 32
  5.3 Static .................................................... 32
  5.4 Method Overloading ....................................... 34
  5.5 Written Exercises ......................................... 34
  5.6 Programming Exercises .................................... 37

6 Arrays 39
  6.1 Motivation ................................................ 39
  6.2 Arrays .................................................... 39
  6.3 Written Exercises ......................................... 40
  6.4 Programming Exercises .................................... 40

7 Searching & Sorting 43
  7.1 Searching ................................................ 43
    7.1.1 Linear Search ....................................... 43
    7.1.2 Binary Search ....................................... 44
  7.2 Sorting ................................................... 45
    7.2.1 Bubble Sort .......................................... 45
    7.2.2 Selection Sort ...................................... 46
    7.2.3 Insertion Sort ...................................... 47
  7.3 Written Exercises ......................................... 48
  7.4 Programming Exercises .................................... 49
Chapter 1
Computers

1.1 Motivation

This chapter concerns basic knowledge of computers. In order to write programs for computers, we need to have a basic understanding of how computers work. Once we have this understanding, we can write programs that make use of the computer’s hardware and software.

A computer is a general-purpose device that can be programmed to carry out a set of arithmetic or logical operations automatically. Since a sequence of operations can be readily changed, the computer can solve more than one kind of problem.

In recent years, computers have become much faster due to advancements in hardware speed and software breakthroughs. However, there is only a limit to how fast these advancements can take us. Therefore, we need to write programs that use the hardware efficiently.

1.2 Introduction to Java/Programming Languages

Java is an object-oriented programming language invented by Sun Microsystems in 1995, and Sun was acquired by Oracle in 2009. One large benefit of Java is that it is run in a virtualmachine, which means that it can be run on almost all major platforms: Windows, Mac OS X, Linux, Solaris, etc.

A programming language is a formal constructed language designed to communicate instructions to a machine; particularly, a computer. Basically, it allows a person to code a sequence of instructions that makes the computer do what he/she wants. A computer program, or just a program, is a sequence of instructions, written to perform a specified task on a computer. A computer requires programs to function, typically executing the program’s instructions in a central processor.

1.3 Java Program Structure

All Java programs follow the same basic structure. A typical Java file (with a “.java” extension) is composed of 1 or more classes, which have 1 or more methods, which have 1 or more statements. We will give more rigor to what these terms mean later. Java classes are (typically) provided in packages, which contain one 1 or more classes.

A typical first program for a programmer in any programming language is called a “Hello, World” program, which prints that phrase to the screen/console. The reason that this is the first program for many programmers is that they can demonstrate that they know the basic structure of a Java program.
To print to the screen, a program (that must be saved in a text file that is called `HelloWorld.java`) to do this is:

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

You must type this program in a raw text editor, not in a rich text editor such as Microsoft Word. The reason for this is that Microsoft Word adds special characters, which are not recognized when you compile and run your program (see below).

For many beginners, this program may be a little daunting and confusing. But as we can see, this program follows the basic structure of a Java program: it has 1 class, named `HelloWorld`; it has one method, named `main`, and it has one statement, which prints to the screen.

For each of the terms, the precise meaning of them will be covered in later chapters. However, the basic structure of:

```java
public class YourProgram {
    public static void main(String[] args) {
        // Insert statements here
    }
}
```

will be enough for most beginning Java programs. As a programmer, you will put the statements between the opening and closing brace of the `main` method.

**Note:** the `main` method is the starting point of execution for every Java application/program.

Note: the text “Insert statements here” is in green. This is a “comment”, in which you can add any text. There are 3 types of Java comments:

- Line comments - start with `//`, and last for the rest of the line.
- Multiline comments - start with `/*` and end with `*/`, and last until the `*/` is reached.
- JavaDoc comments - these have the same form as multiline comments, but have a purpose in providing documentation for Java code. They are typically put right before a method declaration.

## 1.4 Running/Compiling Java Programs

Now that we understand how to create Java programs, we want to run them on our computers. In order to run a Java file, we need to do 2 steps: compile it, and then run it.

A compiler (for most programming languages) is software that translates the user-created code into object (or “machine”) code. Obviously, user-created code is readable by the user, since the user created it. However, object code is not readable, since it is code that the computer can read, and then execute.

There are a few ways of compiling a Java file:

- If you have an Integrated Development Environment (IDE), such as Eclipse, Netbeans, etc. on your machine (they are available on the web for free), running your program is done by
clicking the green arrow ("Run") button in the toolbar. This operation will compile and run the program.

- If you are not using an IDE, and have a Terminal/Command Prompt, etc. application in which you supply commands, you enter these commands to compile and then run a Java program:
  - `javac YourProgramName.java`
  - `java YourProgramName` (Note: no “.java” extension on this command)

- If you do not have a Terminal/Command Prompt application, consult sources online on how to use the “JDK” (which means Java Development Kit).

### 1.5 Compilation Errors

However, if you mistyped during creating your program, you encounter what is called a “compilation error”, which shows an appropriate error message. Common errors include:

- Incorrect class name - the name of the class must exactly match the name of the file that it is saved in.
- Invalid class name - there are rules for what a class name can be. For example, it cannot be a Java keyword (i.e. `public`, `class`; all of the words that highlight in blue).
- No semicolon after statements - all statements must end in a semicolon.
- Not putting quote marks around what is put inside the `System.out.println` statement. This is called a “string” of characters. If no quote marks are put, then the Java compiler cannot recognize what Hello, World means. However, it does know what “Hello, World” means.

### 1.6 Notes

- There are 2 kinds of print statements in Java:
  - `System.out.print` - prints out exactly what is given in the parentheses.
  - `System.out.println` - same as `print` but adds a newline after the text given in the parentheses. The next time something is printed, it will start on the next line.

- There are numerous special characters that appear in between quote marks (called “escape characters”):
  - `\n` - a newline character. Suppose we have this line of code:
    ```java
    System.out.print("Testing newline\nTest2");
    ```
    The output for this code would be:
    ```
    Testing newline
    Test2
    ```
    Notice that the character `\n` was not printed because it has a special purpose: providing a new line. Therefore, one can see that the following lines of code do the same thing:
System.out.println("Text");
System.out.print("Text\n");

- \t - a tab character. The text that follows \t will appear in the same manner as a tab key is used in any other text document.

- " - to print a double quote.

1.7 Written Exercises

1. What does a compiler do?

2. Consider the following Java Program:

```java
public class VendingMachine {
    public static void main(String[] args) {
        System.out.println("Please insert 25c");
    }
}
```

By what name would you save this program on your hard disk?

3. Is Java a functional language, procedural language, object-oriented language, or logic language?

4. What is a plain text file?

5. How is a text file different than a .doc file?

6. What is a source program?

7. What is Java bytecode?

8. What is the program that translates Java bytecode instructions into machine-language instructions?

9. Is Java case-sensitive?

1.8 Programming Exercises

1. Write a program to print the following to the screen using a number of print statements:

```
1 2 3 4 5
2 3 4 5 1
3 4 5 1 2
4 5 1 2 3
5 1 2 3 4
```
Chapter 2

Data Types

Now that we know how to compile and run programs, and print to the screen, we will bump up the complexity of our programs a little. In this chapter, we will be exploring data types, and how to do basic calculations.

2.1 Types

There are 8 “primitive” (fundamental) data types in Java (we will explain constraints that they have later):

- **byte** - an integer with a small range.
- **short** - same as byte, but a larger range.
- **int** - same as short, but even larger range.
- **long** - same as int, but even still larger range.
- **float** - a “real” value with some amount of precision.
- **double** - same as float but with more precision.
- **boolean** - a type that takes one of two possible values: true, or false.
- **char** - a Unicode character. Think of a or b as chars - any single character is a char.

Now, we give the range for each of the primitive data types (all inclusive on both ends). These are all of the values that they can hold:

- **byte**: -128 to +127.
- **short**: -32768 to +32767.
- **int**: $-2^{31}$ to $+2^{31} - 1$.
- **long**: $-2^{63}$ to $+2^{63} - 1$.
- **float**: $1.4 \times 10^{-45}$ to $3.4 \times 10^{38}$.
- **double**: $4.9 \times 10^{-324}$ to $1.7 \times 10^{308}$.
- **boolean**: true or false.
- **char**: 0 to 65535.

Now that we know the 8 fundamental data types, we want to know how to use them. To do that, we have to declare a variable with a type.

2.2 Variable Declaration

Like classes in the previous chapter, declaring variables follow the same basic structure, which is:
For example, if we want a `int` variable called `val` to have a value of 2, we would write the line of code as:

```java
int val = 2; // assign val to a value of 2
```

It’s important to remember a single `=` sign is an assignment, not checking for comparison. Also, we can declare several variables of the same type in the same line using a comma between the names:

```java
int val = 2, otherVal = 3;
```

A few important items to note:

- In a list of declarations, there is no need to declare the type twice (in fact, there will be a compiler error if this is done)

- The convention in Java names is to follow camel casing: the first “word” is lowercase, and the subsequent “words” in the same variable are capitalized (see example above).

Now, there are only certain names that our Java variables can have. The rules are:

- The name cannot be a “keyword” (any of the words in this document that are colored blue, such as `int`, `public`, etc.).

- The first character must be a letter (either upper or lowercase), the underscore symbol, or a dollar sign (`$`). However, a variable name cannot start with a number.

- Any character other than the first may be letters, underscores, the dollar sign, or also numbers.

Here are some variable declarations using all primitive types:

```java
byte b = 1;
short s = 2;
int i = 300;
long l = 1000L; // note the 'L'
float f = 0.56f; // note the 'f'
double d = 523.6; // no suffix
boolean b = true;
char c = 'a'; // chars are surrounded in single quotes
```

After we have initialized a variable, we can also modify it without having to declare it again. Here is an example:

```java
int i = 300;
i = 301; // modifies i's value to be 301
// int i = 301; // this causes a compiler error
    // (can only be declared once)
```

### 2.3 Primitive Operators

Now that we know how to declare and modify values, we need to learn how to do various operations on them, such as addition, subtraction, multiplication, and division. An example showing all of them is the following (with comments):
/* Addition */
// int works the same as byte and short
int i1 = 0, i2 = 3;
int i3 = i1 + i2; // adds i1's value (0) and i2's (3)

// doubles and floats work the same way:
double d1 = 2.5, d2 = 3.0;
float d3 = d1 + d2;
float f1 = 2.5f, f2 = 3.0f;
float f3 = f1 + f2;

// Can add across types, but is promoted to "most precise"
// If we add double + int, the result is a double
int i4 = 4;
double d4 = 5.3;
double d5 = d4 + i4;
// But we can also "cast" (with parentheses) to the type we want if
// it is less "precise" - this truncates the fractional part:
int casted = (int)(d4 + i4); // error without (int)

// cannot do operations on booleans (compiler error)
// boolean b1 = true + false;

/* Subtraction */
// Works with primitive types the same way as addition
int i5 = 0, i6 = 3;
int i7 = i5 - i6; // i6 has value -3

/* Multiplication */
// Multiplication is the same as addition in the same type
// However, we promote to "most precise" value if multiple types
int i8 = 5;
double d6 = 5.3;
double d7 = d6*i8; // promote i8 to a double valued 5.0

/* Division */
// Division is easy to understand except for when the numerator and
denominator are both int/short/byte.
int i9 = 10, i10 = 3;
// What is i9 / i10? We expect 3.33333..., but we get 3 instead.
Java does "integer division", where the numerator and denominator
are both ints, which is truncating all the fractional parts.
int i11 = i9 / i10; // i11 has value 3.
// However, if we "cast" one of the values to a double, then we do
// not have this problem
double d8 = (double)i9 / i10; // cast i9 to a double of value 10.0
// d8 now has value 3.333333...
Another operation is called the mod operator (or “remainder”):

```java
int i1 = 10, i2 = 3;
int i3 = i1 % i2; // i3 has value 1, since 10 remainder 3 = 1.
```

We can also “shorten” times when we are modifying the same value with another. For example:

```java
int i = 5;
i += 3; // equivalent to: i = i + 3;
i -= 3; // equivalent to: i = i - 3;
i *= 3; // equivalent to: i = i * 3;
i /= 3; // equivalent to: i = i / 3;
```

For `int` variables, there are 2 operations that are done very often, so they were put into the language: increment and decrement:

```java
int i = 2;
i++; // equivalent to: i += 1;
i--; // equivalent to: i -= 1;
```

### 2.4 Boolean

The `boolean` type is one that allows us to compare values of variables, and have either a value of `true` or `false`, with their usual interpretations. Some operators that give boolean values are given below, with unneeded parentheses added for clarity:

```java
int i1 = 5, i2 = 6;
boolean larger = (i1 > i2); // false, since 5 < 6
boolean largerOrEqual = (i1 >= i2); // also false
boolean smaller = (i1 < i2); // true
boolean smallerOrEqual = (i1 <= i2); // true
boolean notEqual = (i1 != i2); // true
boolean equal = (i1 == i2); // false
// Note that == means compare value, = means assign
```

There are also operators to work with booleans: “and” (\&\&), “or” (\|\|), and “not” (!).

```java
/* And = && */
boolean t = true, f = false;
boolean andTrueTrue = t && t; // true
boolean andTrueFalse = t && f; // false
boolean andFalseTrue = f && t; // false
boolean andFalseFalse = f && f; // false
/* Or = || */
boolean orTrueTrue = t || t; // true
boolean orTrueFalse = t || f; // true
boolean orFalseTrue = f || t; // true
boolean orFalseFalse = f || f; // false
/* Not = ! */
boolean notTrue = !t; // false
boolean notFalse = !f; // true
```
// We can use multiple operators at once:
boolean myst = !(true && (true || false)); // false

What if we do not want our variable values to change? There is a way, with the final keyword:

final int NUM = 100; // convention: all capital letters for final variables
// NUM++; // compilation error

2.5 String

Now that we fully understand how primitive data types work, we can move on to using classes provided by the Java API (Application Programmer Interface). One such class is provided in the Java package java.lang, called String. This package is so often that Java automatically imports it for you, so the programmer does not have to do extra work.

A String is a set of characters between double quotes. Any character can be inside a string, except for a double quote itself. Examples are:

String str1 = "I love ";
String str2 = "Java";
String str3 = str1 + str2; // is now "I loveJava"
// str3 is called the "concatenation" of str1 and str2
int i1 = 3;
str3 = str1 + i1; // has value "I love3"

However, there is another way to declare a String variable, and that is done using the Java keyword new:

String str1 = new String("I love"); // same as above

String is the only Java class that can declare (also called "instantiate") a variable without the new keyword. All other Java classes must be instantiated with this keyword.

Now we need to see what we can do with these String variables. As we covered in the last chapter, every Java class has one or more methods. The syntax for calling a method is:

Variable.MethodName(Optional Parameters)

Note: we cannot call methods on primitive data types (see later chapters on classes and methods). Examples of various methods in Java for the String class are:

String str1 = "I love Java"
/*String.toUpperCase() and toLowerCase*/
String upper = str1.toUpperCase(); // "I LOVE JAVA"
String lower = str1.toLowerCase(); // "i love java"

/*String.replace(char c, char d)*/
// Replaces all instances of c with d
String repl = str1.replace('a', 'p'); // "I love Jpvp"

// We can even chain method calls
String myst = str1.replace('a', 'p').toUpperCase(); // "I LOVE JPVP"
/* String.length() - int which is # of characters*/
int len = str1.length(); // 11, since there are 11 characters

/* String.charAt(int n) - gets the nth character*/
// Strings are 0-indexed, so the first character is at index 0
// Error at runtime if input >= length()
char firstChar = str1.charAt(0); // has value 'I'
char secondChar = str1.charAt(1); // has value '
char lastChar = str1.charAt(str1.length() - 1); // 'a'

/* String.substring(int n, int m)*/
// gives a String that consists of index n through m, not inclusive on m
String substr1 = str1.substring(0, 3); // "I lo"
String substr2 = str1.substring(2, 6); // "love"

/* String.equals(String other)*/
// gives true (boolean) if the first String is exactly the same as the other
boolean equal1 = substr1.equals(substr2); // false
boolean equal2 = "Equal".equals("Equal"); // true
// We can also check for equality ignoring the case
boolean equal3 = "Equal".equalsIgnoreCase("eqUal"); // true

/* String.indexOf(char c)*/
// Returns index (int) of the first instance of c; otherwise, -1
int found1 = str1.indexOf('o'); // 4
int found2 = str1.indexOf('f'); // -1

/* String.compareTo(String other)*/
// Returns int showing lexicographic ordering of strings according to ASCII table (negative if first < second, 0 if equal, positive if first > second)
int c1 = "A".compareTo("B"); // negative
int c2 = "A".compareTo("b"); // more negative, lowercase has higher ASCII value
int c3 = "A".compareTo("A"); // 0
int c4 = "B".compareTo("A"); // positive

2.6 Scanner

Now that we know how to create and modify String variables, it is important to know how to get input from the user so that we can make our programs usable. For that, we use the Scanner class in the java.util package.

However, since this class is not in the package java.lang, we must import it from the Java library, by putting an import statement at the beginning of our file:
import java.util.Scanner; // import this class from the package
// Equivalent: import java.util.*; // import everything from it
public class SomeClass {
    // ...
}

So how do we use this class? We must initialize it with the new keyword:

Scanner s = new Scanner(System.in); // System.in is the keyboard

This code creates an variable (often called “object” in this case, since it is a class) in which we can call various methods on it to get input from the user. There are various ways of getting input from the user:

Scanner s = new Scanner(System.in);
int x = s.nextInt(); // the first integer entered by the user (after pushing the enter/return key) will be put into x
// If an integer is not entered, then a run-time error will occur.
double y = s.nextDouble(); // if a user enters a double
String str1 = s.next(); // this is the next "word" (no spaces)
String str2 = s.nextLine(); // this is the entire line of characters before pressing the enter/return key

2.7 Math

There is another class, called Math (in the java.lang package) that provides some useful functionalities:

/* Square root */
double num1 = 105.0;
double d1 = Math.sqrt(num1);

/* PI - constant */
double pi = Math.PI;

/* Math.{min, max} - minimum or maximum of 2 ints */
int num1 = 5, num2 = 3;
int max = Math.max(num1, num2); // 5
// also can be: Math.max(num2, num1);
int min = Math.min(num1, num2); // 3

/* Pow - take first double to power of second */
double d2 = 3.5, d3 = 4.3;
double d4 = Math.pow(d2, d3); // = 3.5^4.3

2.8 Written Exercises

1. Give the output of the following program:
2. What will be the output of the following program?

```java
public class Example {
    public static void main(String[] args) {
        String s = new String("Arizona state university");
        char ch1 = s.toLowerCase().toUpperCase().charAt(0);
        char ch2 = s.toUpperCase().charAt(8);
        char ch3 = s.toUpperCase().charAt(s.length() - 1);
        System.out.println("ch 1 is: " + ch1);
        System.out.println("ch 2 is: " + ch2);
        System.out.println("ch 3 is: " + ch3);
    }
}
```

3. What will be the output of the following program?

```java
public class Example {
    public static void main(String[] args) {
        int num1 = 4, num2 = 5;
        System.out.println("4"+"5");
        System.out.println(num1+num2);
        System.out.println("num1"+"num2");
        System.out.println(4+5);
    }
}
```

4. Which of the following correctly invokes the method `length()` of the `String` variable `str` and stores the result in `val` of type `int`?

```java
int val = str.length();
int val = length.str();
int val = length().str;
int val = length(str);
```

5. Evaluate each of the following expressions.

```java
String s = "Programming is Fun";
String t = "Workshop is cool";
System.out.println(s.charAt(0)+t.substring(3, 4));
System.out.println(t.substring(7));
```
6. Evaluate each of the following expressions, assuming \( j \) is an \texttt{int} with value 11, \( k \) is an \texttt{int} with value 3, and \( s \) is a \texttt{String} with value “Ford Rivers”.

\[
\begin{align*}
\text{j} \div \text{k} \\
\text{j} \mod \text{k} \\
\text{s}.\text{substring}(1, 5) \\
\text{s}.\text{length()} \\
\text{s}.\text{charAt}(3)
\end{align*}
\]

7. True or False? The type \texttt{String} is a primitive data type.

8. True or False? The type \texttt{char} is a primitive data type.

9. Write the output of the following program:

```java
public class Question {
    public static void main(String[] args) {
        String str = "hello";
        System.out.println("abcdef".substring(1, 3));
        System.out.println("pizza".length());
        System.out.println(str.replace('h', 'm'));
        System.out.println("hamburger".substring(0, 3));
        System.out.println(str.charAt(1));
        System.out.println(str.equals("hello"));
        System.out.println("pizza".toUpperCase());
        System.out.println(Math.pow(2, 4));
        double num4 = Math.sqrt(16.0);
        System.out.println(num4);
        String s1 = "Clinton, Hillary";
        String s2 = new String("Obama, Barack");
        System.out.println(s1.charAt(s1.length() - 1));
        System.out.println(s2.substring(s2.indexOf(",")+2, s2.length()));
    }
}
```

10. What is the result of \(2/4\) when evaluated in Java? Why?

## 2.9 Programming Exercises

1. Write a Java program that asks the user for the radius of a circle and finds the area of the circle.

2. Write a Java program that prompts the user to enter 2 integers. Print the smaller of the 2 integers.
Chapter 3

Decisions

3.1 Motivation

We now know basic data types and how to use them. Now, we will see how to execute conditional code based on values. We will cover \texttt{if}, \texttt{if-else}, and \texttt{switch} statements.

3.2 If Statements

Suppose we were given a task of printing “True” only when a value entered by the user was larger than another value. Currently, we do not know of a way to do this. Therefore, we have \texttt{if} statements. They have the form:

\begin{verbatim}
if (condition) {···}
\end{verbatim}

where “...” means other lines of code, and \texttt{condition} is evaluated to a \texttt{boolean} value. The rule for \texttt{if} statements is: if the condition evaluates to \texttt{true}, then the code between the braces is executed; if \texttt{false}, then it is not executed. If we have one line of code that will be executed in the \texttt{if} statement, then the braces are optional:

\begin{verbatim}
int x = 8;
if (x > 7)
   System.out.println("This one line");
\end{verbatim}

However, the second line after the \texttt{if} statement in the following example will not execute as if it were in the \texttt{if} statement, even though it is indented:

\begin{verbatim}
int x = 8;
if (x > 7)
   System.out.println("This one line");
   System.out.println("What about this line?");
\end{verbatim}

The above code is functionally equivalent to:

\begin{verbatim}
int x = 8;
if (x > 7) {
   System.out.println("This one line");
}
System.out.println("What about this line?"); // always prints
\end{verbatim}

To fix this, we add braces:
int x = 8;
if (x > 7) {
    System.out.print("This one line");
    System.out.println("What about this line?");
}

and therefore, both lines will print if the condition evaluates to true (which it does in this case).

Now suppose we want to execute some code when the condition is true, and some other code when the condition is false (and only when it is false). Therefore, we need an if-else statement, which is of the form:

    if (condition) { ⋅⋅⋅ } else { ⋅⋅⋅ }

If the condition is false, then the code in the else section will automatically execute, and the code in the if section will not. If the condition is true, then the else section will not execute, and the if section will.

Suppose we have been given a task of printing “true” if the user’s input is greater than or equal to 8, and “false” otherwise. Our code then will look like:

Scanner s = new Scanner(System.in);
int input = s.nextInt();
if (input >= 8) {
    System.out.print("true");
} else {
    System.out.print("false");
}

We can even chain multiple if-else statements together (and even nested ones!):

Scanner s = new Scanner(System.in);
int input = s.nextInt();
if (input >= 8) {
    if (input == 8) {
        System.out.print("Equal to 8");
    } else {
        System.out.print("Larger than 8");
    }
} else if (input >= 4) {
    if (input == 4) {
        System.out.print("Equal to 4");
    } else {
        System.out.print("Larger than 4, < 8");
    }
} else {
    System.out.print("Less than 4");
}

3.3 Switch Statements

An alternative to if-else statements are switch statements, because if-else statements can be very complex and lengthy. A switch statement structure consists of 1 or more case statements
3.3. SWITCH STATEMENTS

(with an optional default case). The programmer supplies a switch statement with a value:

```java
switch (value) {
    // ...
}
```

where value is a primitive type or a String (Note: String is only allowed in later versions of Java). For each of the case statements, value is compared to the value supplied in the case statement (see form below). Only if they are equal will the code after the case statement be executed.

- Each case statement is of the form: case value: /* statements */. At the end of the code section for a case statement, an optional break; statement is allowed. However, not putting this statement at the end of the code section for the case statement will immediately execute the next case statement, regardless of the value.

- An optional default: case statement at the end of the switch block is allowed, which is executed if the input value is not equal to any other case statement’s value.

For example, if we were to print “1” if the input value is equal to 1, and “2” if equal to 2, and ignore any other inputs, our code might look like:

```java
Scanner s = new Scanner(System.in);
int input = s.nextInt();
switch (input) {
    case 1:
        System.out.print("1");
        break;
    case 2:
        System.out.print("2");
        break;
}
```

Now, if we remove the break statements:

```java
Scanner s = new Scanner(System.in);
int input = s.nextInt();
switch (input) {
    case 1:
        System.out.print("1");
    case 2:
        System.out.print("2");
}
```

this code will output “12” if the input is 1, and “2” if the input is 2, which is not the behavior we might want. This is because we removed the break statements, and so the code flows through until it hits a break statement. What a break statement does is exit the switch statement altogether.

On the other hand, if we want to add the requirement of printing “other” if we have any other value than 1 or 2, then we use a default statement:

```java
Scanner s = new Scanner(System.in);
int input = s.nextInt();
switch (input) {
    case 1:
    case 2:
        System.out.print("other");
    default:
        System.out.print("other");
}
```
We can see that this code can be constructed in an *if-else* statement fashion. However, sometimes *switch* and *case* statements are easier to understand and read.

### 3.4 Written Exercises

1. What is the output of the following code?

   ```java
   int depth = 30;
   if (depth >= 29) {
       System.out.println("Bigger than 8!";
       System.out.println("Don't swim!");
   }
   System.out.println("Yes, you can swim.");
   ```

2. What is the output of the following code?

   ```java
   int mystery1 = 12;
   int mystery2 = 42;
   System.out.println("You have: ");
   if (mystery1 >= 8)
       System.out.println("1 ");
   if (mystery2 <= 50 && mystery1 <= 12)
       System.out.println("2 ");
   System.out.println("3.");
   ```

3. If k holds a value of the type *int*, then the value of the expression:

   \[ k \leq 10 \lor k > 10 \]

   a) must be true
   
   b) must be false
   
   c) could be either true or false
   
   d) is a value of type *int*

4. For the following code, fill in the missing condition to check if *str1* and *str2* are the same.

   ```java
   String str1 = "Java is fun";
   String str2 = "Java is fun";
   ```
if ( /* */ )
    System.out.println("String1 and String2 are the same");
else
    System.out.println("String1 and String2 are different");

5. Evaluate the following expressions, assuming that $x = -2$ and $y = 3$.

- $x \leq y$
- $(x < 0) \lor (y < 0)$
- $(x \leq y) \land (x < 0)$
- $((x + y) > 0) \land !(y > 0)$

6. Write the output of the following code:

```java
int grade = 45;
if (grade >= 70)
    System.out.println("passing");
if (grade < 70)
    System.out.println("dubious");
if (grade < 60)
    System.out.println("failing");
```

7. Write the output of the following code:

```java
String option = "A";
if (option.equals("A"))
    System.out.println("addRecord");
if (option.compareTo("A") == 0)
    System.out.println("deleteRecord");
```

8. Write the output of the following code:

```java
double x = -1.5;
if (x < -1.0)
    System.out.println("true");
else
    System.out.println("false");
    System.out.println("after if...else");
```

9. Write the output of the following code:

```java
int j = 8;
double x = -1.5;
if (x >= j)
    System.out.println("x is high");
else
    System.out.println("x is low");
```

10. Write the output of the following code:
double x = -1.5;
if (x <= 0.0) {
    if (x < 0.0)
        System.out.println("neg");
    else
        System.out.println("zero");
} else
    System.out.println("pos");

3.5 Programming Exercises

1. Write a program that asks for 3 integers and prints the median value of the 3 integers, using only if statements.

2. Write code that ensures that an int variable called number is an odd integer.
Chapter 4

Loops

4.1 Motivation

With the last chapter, we now know how to execute conditional code. This chapter will greatly increase our powers in what we can do with programming, by being able to execute code many times without having to re-write (or copy and paste) code over and over.

4.2 Introduction to Loops

Loops are a way of executing some code an arbitrary number of times. What determines the number of times a loop executes is based largely on a condition, just like if and switch statements were. There are 3 kinds of loops in Java:

- while loops
- do-while loops
- for loops

4.3 While loops

while loops are the simplest to understand kind of loop. The structure of one is:

```java
while (condition) {
    ···
}
```

The semantics are: the loop will continue as long as the condition evaluates to true. The main reason while loops are used is that we may not know the number of times looping occurs. If we do, we usually use a for loop (see below). However, both while and for loops can be converted back and forth.

For example, let’s say we are given a task of displaying a positive integer that the user enters, and continue doing so until the user enters 0. Setting up is easy:

```java
Scanner s = new Scanner(System.in);
while(/* something */) {
    // somehow take user input and display it
}
```

Let’s initialize an int variable, initialized to the first input a user gives:
Scanner s = new Scanner(System.in);
System.out.println("Enter an integer: ");
int input = s.nextInt();
while(/* something */) {
    // somehow take user input and display it
}

Now, we need to create the condition for the while loop. We stop after the user enters 0, so let’s make that condition (and display it):

Scanner s = new Scanner(System.in);
System.out.println("Enter an integer: ");
int input = s.nextInt();
while (input != 0) {
    System.out.println(input);
    // anything else go here?
}

This code seems good, but unfortunately we can encounter what is called an “infinite loop”, which does not stop looping. If we enter a non-zero integer, then we print that output. Then since there is nothing else to do in the loop body, we check the condition again, which evaluates to true, prints out, loops back, prints out, etc. We need to get the user input again at the end of the loop:

Scanner s = new Scanner(System.in);
System.out.println("Enter an integer: ");
int input = s.nextInt();
while (input != 0) {
    System.out.println(input);
    System.out.println("Enter another integer: ");
    input = s.nextInt(); // avoid infinite loop
}

Now our solution code for the example is correct.

4.4 Do-While loops

do-while loops are the same as while loops, but guarantee that the loop body will be executed at least once. Remember our earlier while loop example: if we initially entered 0, then the loop’s condition evaluates to false, and does not execute the body. Therefore, it loops 0 times. The structure of a do-while loop is:

do { ... } while (condition);

Note: the semicolon after the last parentheses of the while’s condition is necessary. For example, if we modify our example from above to be a do-while loop, we guarantee that the user must enter at least one integer:

Scanner s = new Scanner(System.in);
System.out.println("Enter an integer: ");
int input = s.nextInt();
do {
    System.out.println(input);
4.5. FOR LOOPS

System.out.println("Enter another integer: ");
input = s.nextInt(); // avoid infinite loop
} while (input != 0);

4.5 For loops

For loops are used when we know precisely how many times the loop will execute. However, they are equivalent in functionality to while loops (i.e. one can construct a for loop from a while loop, and vice versa). All for loops have the same structure:

for (initialization; condition; modification) { ··· }

- Initialization - creating a variable, usually called the “loop control” variable.
- Condition - checking a condition, usually related to the loop control variable, against some value. The loop will continue to iterate (and do the modification step) as long as the condition is true.
- Modification - modify, usually the loop control variable, at each iteration of the loop.

For example, let’s say we are given the problem of summing all of the integers from 1 to \( n \), where \( n \) is an integer \( \geq 1 \). An example solving this problem is the following:

```java
Scanner s = new Scanner(System.in);
int n = s.nextInt();
int sum = 0;
// put some kind of for loop here
System.out.println(sum);
```

Now we need to reason how to construct our for loop. Let’s initialize a loop control variable, called \( i \), to be 1. Then, for each iteration of the loop, add \( i \) to \( sum \), and at each iteration, add 1 to \( i \). An “unraveling” of the loop’s logic will be this:

- \( sum = 0 \).
- 1st iteration of loop: \( i = 1, \) add to \( sum, \) \( sum = 1, \) add 1 to \( i (i = 2) \).
- 2nd iteration of loop: \( i = 2, \) add to \( sum, \) \( sum = 3, \) add 1 to \( i (i = 3) \).
- 3rd iteration of loop: \( i = 3, \) add to \( sum, \) \( sum = 6, \) add 1 to \( i (i = 4) \).

and so on. By this logic, we want the loop to stop at \( n \), inclusive. So, we need to have a condition of \( i \leq n \). And on each loop, we execute \( i++ \). Therefore, our final code will be:

```java
Scanner s = new Scanner(System.in);
int n = s.nextInt();
int sum = 0;
for (int i=1; i<=n; i++) {
    sum += i; // add to sum
}
System.out.println(sum);
4.6 Loop Example

In many programming interviews, there are questions designed to test one’s knowledge of loops. One of them is called “FizzBuzz.” The idea is this: print “Fizz” if the number is divisible by 3, “Buzz” if the number is divisible by 5, “FizzBuzz” if the number is divisible by 3 and 5 (i.e., 15), and nothing otherwise. Now, we want the user to input an integer \( n \), and from 1 to \( n \), we want to print the appropriate value. We start with a setup of our program:

```java
public class FizzBuzz {
    public static void main(String[] args) {
        // something goes here...
    }
}
```

Now we need to add appropriate code to get an integer \( n \) from the user:

```java
import java.util.Scanner;
public class FizzBuzz {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        int n = s.nextInt(); // get the integer
    }
}
```

Since we know the number of loop iterations (i.e., \( n \)), we can just use a for loop to iterate from 1 to \( n \):

```java
import java.util.Scanner;
public class FizzBuzz {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        int n = s.nextInt(); // get the integer
        for (int i = 1; i <= n; i++) {
            // what else?
        }
    }
}
```

Now we just need to insert an if and else statements to check if \( n \) is divisible by 3, 5, or 15 (and then print the appropriate value). However, we cannot just do this:

```java
import java.util.Scanner;
public class FizzBuzz {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        int n = s.nextInt(); // get the integer
        for (int i = 1; i <= n; i++) {
            if (i % 3 == 0) {
                System.out.println("Fizz");
            } else if (i % 5 == 0) {
                System.out.println("Buzz");
            } else if (i % 15 == 0) {
```
System.out.println("FizzBuzz");
}
}
}

The last clause will never be reached. If i is divisible by 15, then it is divisible by 3 or 5 as well. Therefore, it would have went inside one of the other conditionals. Therefore, we just need to move it to be the first conditional:

```java
import java.util.Scanner;
public class FizzBuzz {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        int n = s.nextInt(); // get the integer
        for (int i = 1; i <= n; i++) {
            if (i % 15 == 0) {
                System.out.println("FizzBuzz");
            } else if (i % 3 == 0) {
                System.out.println("Fizz");
            } else if (i % 5 == 0) {
                System.out.println("Buzz");
            }
        }
    }
}
```

There are other ways to do this - for example, we can just have the last 2 conditionals, but use `print` instead of `println`, and then add a newline afterwards.

### 4.7 Written Exercises

1. What are the 3 kinds of loops in Java?

2. What is the output of the following loop? How many times does the loop execute?
```java
int n = 979;
for (int j = 0; j <= n; j++) {
    System.out.println("Hello");
}
```

3. What is the output of the following loop? How many times does the loop execute?
```java
int j = 1;
int n = 5;
while (j <= n) {
    System.out.println("Hello");
    n--;
}
```

4. What is the output of the following loop? How many times does the loop execute?
int n = 5;
for (int j = 1; j <= n; j += 3) {
    System.out.print("Hello ");
    int k = j;
    while (k < n) {
        System.out.println("Good Morning");
        k++;
    }
    j--;
}

5. What is the output of the following code?

String name = "Richard M. Nixon";
boolean startWord = true;
for (int i = 0; i < name.length(); i++) {
    if (startWord)
        System.out.println(name.charAt(i));
    if (name.charAt(i) == ' ')
        startWord = true;
    else
        startWord = false;
}

6. What is the output of the following loop? How many times does the loop execute?

int j = 1;
while (j <= 11) {
    System.out.println("Hello ");
    j = j + 3;
}

7. What is the output of the following code?

int n = 1, i = 1;
while (i < 7) {
    n = n * i;
    i += 2;
}
System.out.print(n);

4.8 Programming Exercises

1. Write a loop that reads in int values until the user enters 0 and prints out how many values entered are greater than 10.

2. Write a loop that will print out every other letter in a String variable called str. For example, if the String was “Hello There”, then “HloTee” will be printed.
Chapter 5

Introduction to Classes

Now that we understand how loops work, we want to be able to create our own classes! Just like the ones that we have been using, such as Scanner, String, Math, and others.

5.1 Class Structure

A class follows the same format as we have been for our main programs. However, there does not have to be a main method in a class. The only requirement for main is that the class we run a program from contains a main method.

A class contains what are called “instance variables”, a constructor, and 1 or more methods. But before we can talk about these, we need to cover visibility modifiers.

5.1.1 Visibility Modifiers

We have been using public when we create our programs so far. Now we will cover what this means, and what other visibility modifiers are. They are, for any variable/method/class:

- public - any class can see it.
- private - only the class that contains it can see it.
- protected - only the class that contains it and any children that inherit from it can see it.

Do not worry about this visibility modifier - we will not use it in this chapter.

5.1.2 Instance Variables

Instance variables are variables that exist inside a class when they are created. We usually write instance variables to have a visibility of private, to enforce what is called “encapsulation”, which means that we hide data (the instance variables) that we do not want “outsiders” to change.

For example, let’s create a BankAccount class:

```java
public class BankAccount {

}
```

In order to make an instance of the class, we need to create a “constructor” for the class, which executes code inside it when we use the new operator on the class, in the exact same way that we have used it for other Java-provided classes. Notes about the constructor:
• No return type
• Optional parameters
• Any number of constructors allowed as long as they differ in signature (number of arguments or different types of arguments).
• The default constructor—with no arguments—is provided by Java if none is provided by the programmer.

For example, let’s modify our BankAccount class that has the name of the person with the account, and a starting balance. When we initialize our BankAccount object in the main method, we will use the new operator, which passes in the values supplied to name and startAmount below:

```java
public class BankAccount {
    // visibility of public - need to see outside of class
    public BankAccount(String name, double startAmount) {
        // optional: use this operator
        // this.personName = name;
        // this references the BankAccount class when inside the class, and what comes after the dot references the instance variable with that name
    }
}
```

Now what do we do with these parameters? We need a way of “saving” them inside the class for later use. That is where instance variables come in. Let’s create our instance variables with public visibility for now:

```java
public class BankAccount {
    public String personName;
    public double balance;

    // visibility of public - need to see outside of class
    public BankAccount(String name, double startAmount) {
        personName = name; // set name
        balance = startAmount; // set amount
        // this.personName = name;
    }
}
```

And in our main (or other that instantiates a BankAccount object) method:

```java
public class BankAccountTest {
    public static void main(String[] args) {
        BankAccount b = new BankAccount("Jane", 0.0);
    }
}
```

Since we have declared our instance variables as public, we can modify the corresponding values of our instance to anything we want by using the “dot” operator, in the same way we use Math.PI on the Math class:
public class BankAccountTest {
    public static void main(String[] args) {
        BankAccount b = new BankAccount("Jane", 0.0);
        b.personName = "Another Name";
        b.balance = 500000;
    }
}

We can clearly see why this is not preferred. Therefore, we need to mark the visibility of our instance variables as private (and the operations above will result in a compile-time error):

public class BankAccount {
    private String personName;
    private double balance;

    public BankAccount(String name, double startAmount) {
        personName = name; // set name
        balance = startAmount; // set amount
    }
}

This is a good start, but we want to be able to deposit or withdraw some amount into balance. Therefore, we need to use “setters” (also called “mutators”) and “getters” (also called “accessors”). Setters are about changing some internal value to a passed-in value, and getters are about getting internal values back to the user:

public class BankAccount {
    private String personName;
    private double balance;

    public BankAccount(String name, double startAmount) {
        personName = name; // set name
        balance = startAmount; // set amount
    }

    /*
    Getters have the form:
    public <Type> get<NameOfVar>() {
        return <NameOfVar>;
    }
    */
    // get the current balance
    public double getBalance() {
        return balance;
    }

    /*
    Setters have the form:
    public void set<NameOfVar>(<Type> other) {
        <Instance Var> = other;
    }
    */
But this introduces the same problem as earlier - we can make our balance be any value we want. Therefore, we should get rid of the setter, and introduce a `deposit` method, which adds to the balance with the input. We can leave the getter because it does not modify the value of `balance`.

```java
public class BankAccount {
    private String personName;
    private double balance;

    public BankAccount(String name, double startAmount) {
        personName = name; // set name
        balance = startAmount; // set amount
    }

    // get the current balance
    public double getBalance() {
        return balance;
    }

    public void deposit(double amount) {
        balance += amount;
    }
}
```

And in main, we can use these methods accordingly:

```java
public class BankAccountTest {
    public static void main(String[] args) {
        BankAccount b = new BankAccount("Jane", 0.0);
        b.deposit(50.0);
        System.out.print(b.getBalance());
    }
}
```

Note: a method (terminology to be covered later) called `toString`, which returns a `String`, is commonly implemented among programmer-created classes. The returned `String` is a textual description of the state of the class and instance variables. For instance, we could use our `BankAccount` class and create a `toString` method that returns:

Name: Jane
Balance: 50.0

with the following code:

```java
public String toString() {
    // return textual description
    String toReturn = "";
    toReturn += "Name: " + personName;
    toReturn += "\n";
    return toReturn;
}
```
5.2. METHODS

```java
toReturn += "Balance: " + balance;
return toReturn;
}
```

And when the object is printed out, even without calling `toString()`, the textual description will be printed:

```java
// in main
BankAccount b = ...;
System.out.print(b); // calls toString() automatically
// equivalent to:
System.out.print(b.toString());
```

**Note**: Not providing a `toString()` method in a class, when printing an instance, will print a hash value (i.e. random letters and numbers) instead of what is expected, along with the name of the class.

5.2 Methods

We have seen various “methods”, but we want to improve on the functionality of our classes that we create. Therefore, we use methods to act on the objects we create.

5.2.1 Structure

The typical structure of a method is:

```java
⟨Visibility Modifier⟩⟨Return Type⟩⟨Method Name⟩⟨(Optional Parameters)⟩{···}
```

For example, if I want to create a method called `test` which takes a `String` parameter and returns nothing (i.e., `void`), we would probably write it like this:

```java
public void test(String param) {
    // ...
}
```

Let’s say that this method is in a class called `A`. Therefore, to call the method on the class, we would write (in `main`):

```java
A a = new A(); // or some other constructor
a.test("Hello");
```

We say that we “call” the method `test` on the object `a` of type `A`. We will get a compilation error if instead of `public`, we make the visibility modifier be `private`. Also, a method can only have one return type, or `void`, which means “not returning a value,” but is nevertheless a return type (just not a value). However, if we put `void` as our return type, we can have a `return` statement to immediately exit from the method:

```java
public void someMethod() {
    for (int i=0; i < 100; i++) {
        if (i == 57) {
            return; // will immediately stop execution of method
            // return 5; // compilation error, since it is void
        }
    }
```
5.2.2 Methods Returning Values

Methods can “return” types, with the `return` keyword. For example, if we want to implement a method that takes in 2 `int` parameters and returns their product, we can do this:

```java
public int multiply(int num1, int num2) {
    return num1 * num2; // return this value
}
```

We can use this method, say in a class called C, to call the method and use the value that is returned:

```java
C c = new C();
int result = c.multiply(3, 4); // result is 12
```

Since we specified `int` as the return type of `multiply`, we must return a `int` at some point in the method. We will have a compilation error if we don’t return a type when we say there will be, or returning an incorrect type. Also, we must have a `return` statement at all points that are reachable within the program (or combine them). For example, here is a method that will cause a compilation error:

```java
public int multiply(int num1, int num2) {
    if (num1 > 0) {
        System.out.print("num1 > 0");
        // compiler error here because no return value
    } else {
        return num1 * num2; // return this value
    }
}
```

We can see that if `num1 > 0`, then we print that statement out, do not execute the `else` clause, and continue. Since we can have a way through the method without returning, it causes a compilation error. We can mitigate this by adding a `return` statement either inside the `if` statement or at the end of the method (the “catch-all” case):

```java
public int multiply(int num1, int num2) {
    if (num1 > 0) {
        System.out.print("num1 > 0");
        return 0; // fixed compilation error
    } else {
        return num1 * num2; // return this value
    }
    // can also put here: return 0;
}
```

5.3 Static

Now that we have created instance methods, we want to create class methods, which do not involve any instance state. The keyword `static` in Java is meant to mean “only one of these exists”. If I
5.3. STATIC

have a **static** variable in a class (in the same place an instance variable will be), there will only be one copy of that variable, regardless of how many instances there are of that class.

For example, let’s say we have a class called `B` that has a **static** field called `numInstances` of type `int` which is incremented every time the constructor is called. We would design the code like this:

```java
public class B {
    private static int numInstances = 0; // start with a value
    public B() {
        numInstances ++;
    }
    public int getNumInstances () {
        return numInstances;
    }
}
```

If we call the following code from `main`, we get this output:

```java
B b1 = new B();
System.out.println(b1.getNumInstances()); // prints 1
B b2 = new B();
System.out.println(b2.getNumInstances()); // prints 2
```

Now if we change the code to not have the instance variable be static, then we have:

```java
public class B {
    private int numInstances = 0; // remove "static"
    public B() {
        numInstances ++;
    }
    public int getNumInstances () {
        return numInstances;
    }
}
```

And the outputs are:

```java
B b1 = new B();
System.out.println(b1.getNumInstances()); // prints 1
B b2 = new B();
System.out.println(b2.getNumInstances()); // prints 1
```

which is not the behavior that we want. Now we can get a larger understanding of various conventions we have been using:

- The main method’s use of `static` in its signature shows that only 1 copy of that method can exist at a time in a Java program. Therefore, there is no ambiguity as to where execution starts.

- For other code, such as `Math.PI`, we can have an understanding as to possibly how it and some of its static methods are implemented:

```java
public class Math {
    // ...
```
5.4 Method Overloading

Now that we know how to make instance and class methods, we want to be able to have multiple methods of the same name but take different arguments. This process is called “method overloading”. Method overloading happens when the following criteria are met (assume 2 methods):

- The methods are in the same class
- The methods have the same return type and method name
- The methods have a different number or type of parameters (a different “method signature”)

For example, if we want to have a method called print and want to print “double” if the input is a double, and “int” if the input is an int, we probably will design our code like this:

```java
public void print(double d) {
    System.out.print("double");
}
// overloaded method:
public void print(int i) {
    System.out.print("int");
}
```

However, if we have multiple methods with the same name, same parameters, but a different type, we have a compiler error. The reason for this is that there can be ambiguity in which method to choose. For example, the following methods will cause a compilation error:

```java
public int method() {
    return 0;
}
// overloaded method, causes compilation error:
public long method() {
    return 0;
}
```

We can see that method overloading is generally used for providing different functionality to different types of inputs, whether that be the number of them, or their types.

5.5 Written Exercises

1. Which of the following enforces Encapsulation?
   (a) Make instance variables private
5.5. **WRITTEN EXERCISES**

(b) Make methods public
(c) Make the class final
(d) Both a and b
(e) All of the above

2. Use the following class to answer the questions below:

```java
public class Store {
    private int quantity;
    private double price;
    public Store(int q, double p) {
        quantity = q;
        price = p;
    }
    public int getQuantity() {
        return quantity;
    }
    public void setPrice(double p) {
        price = p;
    }
    public double calcTotal() {
        return price * quantity;
    }
}
```

(a) What is the name of the class?
(b) List all instance variables of the class.
(c) List all methods of the class.
(d) List all mutators in the class.
(e) List all accessors in the class.
(f) List which method is the constructor.

3. True or False? If no constructor is provided, then Java automatically provides a default constructor.

4. True or False? A method must have at least 1 return statement.

5. Write the output generated by the following program:

```java
public class Two {
    private double real, imag;
    public Two(double initReal, double initImag) {
        real = initReal;
        imag = initImag;
    }
    public double getReal() {
        return real;
    }
    public double getImag() {
        return imag;
    }
}
```
public Two mystery(Two rhs) {
    Two temp = new Two(getReal()+rhs.getReal(),
        getImag()+rhs.getImag());
    return temp;
}

public class Test {
    public static void main(String[] args) {
        Two a = new Two(1.2, 3.4);
        Two b = a.mystery(a);
        Two c = b.mystery(b);
        System.out.println("1. " + a.getReal());
        System.out.println("2. " + a.getImag());
        System.out.println("3. " + b.getReal());
        System.out.println("4. " + b.getImag());
        System.out.println("5. " + c.getImag());
    }
}

6. What is a static variable? What is a static method?

7. Using the code below, how many copies of the variable number exist after instantiating 374 different AmazingClass objects?

    public class AmazingClass {
    private static int number;
    public AmazingClass(int a) {
        number = a;
    }
    public int twice() {
        number *= 2;
        return number;
    }
    }

8. Using the code from above, what is the value of number after each of the following statements? (For each line below, assume the preceding ones have already been executed).

    AmazingClass ac1 = new AmazingClass(3);
    AmazingClass ac2 = new AmazingClass(7);
    ac1.twice();
    ac2.twice();

9. What is method overloading?

10. What are the valid method headings assuming they are written in the same class?

    public void v()
    public double v()
public double f2()
public double f2(int d)

public double sum(int left, int right)
public int sum(int left, int right)

public String s(int n)
public int S(int n)

5.6 Programming Exercises

1. For the Store class in the Written Exercises above, do the following:
   a) Write a mutator for the quantity.
   b) Write an accessor for the price.
   c) Write a line of code that will create an instance called videoStore that has quantity 100 and a price of 5.99.
   d) Call the calcTotal method with the videoStore object (from part c) to print out the total.

2. Correct the following class definition if you think it will not work:
   public class Student {
      private String name, major;
      public Student() {
         name = "???";
         major = "xxx";
      }
      public Student(String n, String m) {
         n = name;
         m = major;
      }
      public String getMajor() {
         return m;
      }
      public String getName() {
         return n;
      }
   }

3. Write a boolean method called allDifferent that takes 3 int numbers and returns true if they are all different, and false otherwise.

4. Write a boolean method called isPrime that takes in an int number and returns true if it is prime, and false otherwise.
5. Implement a class called `AsuStudent`. The class should keep track of the student’s name, number of classes registered, hours spent per week for a class (consider a student devotes the same amount of time for each of his/her classes per week). Implement a `toString` method to show the name and number of classes registered by a student, a `getName` method to return the name of the student, a `getTotalHours` method to return the total number of hours per week, and a `setHours` method to set the number of hours the student devotes for each class.
Chapter 6

Arrays

6.1 Motivation

Arrays are a very powerful tool in a programmer’s arsenal. They allow managing many variables at once without much hassle.

6.2 Arrays

The typical structure for creating an array is:

\[
\langle \text{Type}\rangle\langle \text{Name}\rangle = \text{new } \langle \text{Type}\rangle[\langle \text{Integer}\rangle];
\]

where \(\langle \text{Integer}\rangle\) is an integer greater than 0. For example, if we want to create an array of \text{int} variables with 10 elements, we can construct it like this:

\[
\text{int}[] \text{array} = \text{new int}[10]; \quad // \text{10 elements}
\]

Now how do we access or modify elements of an array? Here are some code examples on how to do so:

\[
\text{int}[] \text{array} = \text{new int}[100]; \quad // \text{100 elements}
\]

// By default, each element is 0 (primitives) or null (objects)
// indexes are same as in String, etc. 0 – 99 indexes in this array
array[0] = 10; // set first element to 10
array[1] = 9; // set second element to 9
array[99] = 255; // set last element to 255
// array[100] = 100; // ArrayOutOfBoundsException, because
// index is larger than the highest addressable one
// array[-1] = -1; // Also an ArrayOutOfBoundsException

// Cannot assign/get different types or non-integer indexes:
// String s = array[10]; // compiler error
// array[10] = new String(); // compiler error
// System.out.println(array[2.5]); // compiler error

// However there is a ".length" attribute we can use:
System.out.println(array.length); // 100
Therefore, we can set to the last element without a hardcoded number:
array[array.length-1] = 100;
// Getting an element is the same:
int n = array[56]; // get element at index 56
// We can loop over elements:
for(int i = 0; i < array.length; i++) {
    System.out.print(array[i] + " ");
}
// Or equivalently, the range-based for loop:
for(int i : array) { // "for each i in array"
    System.out.print(i + " ");
}

6.3 Written Exercises

1. What are the indices for the first and last positions of any array x?

2. Immediately after instantiating a new array of primitives (ints, doubles, etc.), what fills the array? What about an array of objects?

3. What happens when you try to access an array element past the end of the array?

4. Use the following array x to answer the following questions:
4 8 5 1 6 3 2

   (a) What value is given by x[1]?
   (b) What value is given by x[6]?
   (c) What value is given by x[7]?
   (d) What value is given by x.length?

6.4 Programming Exercises

1. Instantiate three arrays called x, y, and z of type int, String, and BankAccount (respectively), all of size 10.

2. Write a for-loop to sum all of the elements of an array x of type int.

3. Write a for-loop to multiply by 2 each element in an array x of type int.

4. Write code to store the largest number in an int array x into a variable called max.

5. Write code to count how many numbers in an array x are strictly larger than 4, and store that total in a variable called total.

6. Write code to print out every other element in an array x separated by tabs.
7. Write code to shift each number in an array \( x \) one place to the right (Note: there will be 2 copies of the 1st element when the code finishes).

8. Write code to print the contents of an array \( x \) in reverse order, one element for each line.

9. Write a method called \texttt{append} that appends the two \texttt{int} arrays passed as arguments and returns an array of type \texttt{int} as the result. For example, if the first array argument was \{1, 2, 3\}, and the second was \{4, 5, 6, 7\}, append returns \{1, 2, 3, 4, 5, 6, 7\} (Hint: what is the size of the array to be returned?).

10. Write a method called \texttt{findMin} that returns the smallest element in an \texttt{int} array that is passed as an argument. For example, if the array was \{4, 7, 9, 12, 8, 1, 5\}, the method would return 1 (Hint: use an earlier exercise for this).
Chapter 7

Searching & Sorting

7.1 Searching

Now that we know how to make arrays, we want to use them efficiently. This chapter and the next will be a more “high-level” approach to problems in programming and Computer Science. Searching is a well-studied and important problem. We cover 2 basic searching algorithms that will find elements in an array quickly and efficiently: linear and binary searching.

7.1.1 Linear Search

Linear Search is searching from one end of the array until the end, and stopping if we find the element that we desire. Pseudocode to match a linear search is:

```
function LinearSearch(array a, variable v):
    for(item i in a):
        if i == v:
            return i; // or any non-"error" value (in array)
        return -1; // some "error" value (not in array)
```

For example, if we have an array of int variables and we are searching for one, we can construct code such as this:

```java
public boolean linearSearch ( int [] array, int search ) {  
    for ( int i : array ) {  
        if (i == search) {  
            return true;  
        }  
    }  
    return false;  
}
```

However, this loop can be inefficient, because we have no guarantee what the ordering or contents of array are. We have to search (on average) $\frac{n}{2}$ elements. Therefore, we cannot make a “guess” as to improvements can be done.

However, if we can have a guarantee on the ordering of elements in the array, we can get better performance, as shown by binary search.
7.1.2 Binary Search

We can improve our algorithm by using binary search, on one condition: we must guarantee that the input array is already sorted. We will cover how to sort an unsorted array later. Pseudocode of the algorithm is the following:

```plaintext
function BinarySearch(array a, int key):
    low = 0, high = a.length;
    while high >= low:
        mid = average(low, high);
        if a[mid] == key:
            return mid; // found
        else if a[mid] < key:
            low = mid + 1;
        else
            high = mid - 1;
    return -1; // not found
```

The algorithm works like this: since the array is sorted, we check the middle value. If the value to search is larger than that middle value, we “cut off” the lower half of the array, since we can guarantee it is not in that part (and likewise for less than). Then we check the middle half of the top half of the array, and keep adjusting what is the “low” and “high” ranges of what part of the array we are searching. We stop when we keep adjusting “low” to the point that it is higher than “high” (it does not make sense to search in a range that is backwards). At that point, we have not found our element, and we return.

For example, we can create a similar procedure in Java with an int array (the concept works with Strings also, just comparing their lexicographic ordering instead):

```java
public int binarySearch(int[] array, int k) {
    int min = 0, max = array.length - 1;
    while (max >= min) {
        int mid = (min + max) / 2;
        if (a[mid] == k) {
            return mid;
        } else if (a[mid] < k) {
            min = mid + 1;
        } else {
            max = mid - 1;
        }
    }
    return -1; // not found
}
```

Now for an example. Suppose we have an array consisting of 
{-3, -1, 2, 4, 7}. Clearly, the array is sorted, so we can use binary search. We want to search for the element 4. Here is the execution of binary search:

```plaintext
// Start binary search
min = 0, max = 5
// 7 >= -3, so enter while loop
mid = (0+5)/2 = 2 (integer division)
a[mid] = 2
```
7.2 Sorting

Another problem in Computer Science is sorting data. For binary search in the last chapter, we required that the data be sorted. But how do we sort an arbitrary array? This chapter covers various ways to sort. One thing to keep in mind is this: all array types can be sorted.

7.2.1 Bubble Sort

Bubble sort is an inefficient, but easy to understand, way of sorting data. Almost all sorting algorithms, in the worst case, take \( \approx n \times \log(n) \) operations, where \( n \) is the length of the array. Bubble sort takes \( \approx n^2 \) operations. It is never used in production code, but is essential for understanding how to sort data. The pseudocode for the algorithm is:

```plaintext
function BubbleSort(array a):
    n = a.length;
    swapped = true;
    while swapped == true:
        swapped = false;
        for i=1 to n-1:
            if a[i-1] > a[i]
                swap(a[i-1], a[i]);
                swapped = true;
```

The reasoning is this: start with the first 2 elements. If the first is larger than the second, swap them. Then move onto the 2nd and 3rd items, and swap if necessary, etc. until the last 2 elements. If we did a swap operation at any point, \( \text{swapped} \) is \( \text{true} \), and we keep repeating the outer \( \text{while} \) loop. We set \( \text{swapped} \) to \( \text{false} \) on every iteration of the \( \text{while} \) loop. The only way \( \text{swapped} \) can be \( \text{false} \) is if we do not set it to \( \text{true} \), which means that the data is sorted (each \( i \)-th element is larger than the \( i-1 \)-th element). Therefore, we are done.

For example, let’s assume we have the array \{5,3,1,4,−2\}. Let’s see each step of running the bubble sort algorithm:

```
5 3 1 4 -2 // original
```
// enter for loop - a[0] > a[1], so swap:
3 5 1 4 -2
// a[1] > a[2], so swap:
3 1 5 4 -2 // etc:
3 1 4 5 -2
3 1 4 -2 5

// completed for loop - swapped = true, so repeat:
3 1 4 -2 5 // unmodified
1 3 4 -2 5
1 3 4 -2 5 // still checking:
1 3 -2 4 5

// completed for loop - swapped = true, so repeat:
1 3 -2 4 5 // unmodified
1 3 -2 4 5
1 -2 3 4 5
1 -2 3 4 5

// completed for loop - swapped = true, so repeat:
1 -2 3 4 5 // unmodified
-2 1 3 4 5
-2 1 3 4 5
-2 1 3 4 5

// completed for loop - swapped = true, so repeat:
-2 1 3 4 5 // unmodified
-2 1 3 4 5
-2 1 3 4 5
-2 1 3 4 5

// completed for loop - swapped = false, so we are done.

We can see why this algorithm can take a long time to sort.

### 7.2.2 Selection Sort

Selection Sort is an improvement on Bubble Sort in terms of number of operations done to sort a list. The pseudocode is:

```plaintext
function SelectionSort(array a):
    for j=0 to n-1:
        min = j;
        for i=j+1 to n-1:
            if a[i] < a[min]:
                min = i;
        if min != j:
            swap(a[j], a[min]);
```
It is best to see how Selection Sort works with an example. Let’s say we had the same array as before: \{5,3,1,4,-2\}:

\[
\begin{align*}
5 & \quad 3 & \quad 1 & \quad 4 & \quad -2 & \quad // \text{ unmodified} \\
& \quad \text{min} = 0, \text{ enter 2nd for loop} \\
& \quad a[1] > a[0], \text{ so skip if statement} \\
& \quad a[2] < a[0], \text{ so set min = 2} \\
& \quad a[3] > a[2], \text{ so skip if statement} \\
& \quad a[4] < a[2], \text{ so set min = 4} \\
& \quad \text{since 4 != 0, swap a[0] with a[4]}: \\
& \quad -2 & \quad 3 & \quad 1 & \quad 4 & \quad 5 \\
\end{align*}
\]

\[
\begin{align*}
& \quad // \text{ 2nd iteration:} \\
& \quad \text{min} = 1, \text{ enter 2nd for loop} \\
& \quad a[2] < a[1], \text{ so set min = 2} \\
& \quad a[3] > a[2], \text{ so skip if statement} \\
& \quad a[4] > a[2], \text{ so skip if statement} \\
& \quad \text{since 2 != 1, swap a[1] with a[2]}: \\
& \quad -2 & \quad 1 & \quad 3 & \quad 4 & \quad 5 \\
& \quad \text{// etc.}
\end{align*}
\]

We can see that on each iteration of the outside loop, we select the smallest of the “remaining” items in the array. Then, if the smallest is not the current value, then we swap. This has many fewer swap operations than Bubble Sort.

### 7.2.3 Insertion Sort

Insertion Sort has roughly the same running time as Selection Sort, but is another approach to sorting. The pseudocode is:

```
function InsertionSort(array a):
    for i=1 to a.length:
        j = i;
        while j > 0 and a[j-1] > a[j]:
            swap(a[j], a[j-1]);
            j--;
```

Again, let’s use our example of \{5,3,1,4,-2\} as our array:

\[
\begin{align*}
5 & \quad 3 & \quad 1 & \quad 4 & \quad -2 & \quad // \text{ unmodified} \\
& \quad \text{begin for loop, i=1, j=1} \\
& \quad \text{begin while loop:} \\
& \quad \text{j > 0 and a[0] > a[1]}: \\
& \quad 3 & \quad 5 & \quad 1 & \quad 4 & \quad -2 \\
& \quad \text{exit for loop}
\end{align*}
\]

\[
\begin{align*}
& \quad // \text{ 2nd iteration, i=2, j=2} \\
& \quad \text{begin while loop:} \\
& \quad \text{j > 0 and a[1] > a[2]}:
\end{align*}
\]
What the algorithm does is: on the first iteration, the first element is sorted (trivially). Next, look at the second element, and swap it left until it is in its “sorted” position (i.e. now the first 2 elements are sorted). Now, do the same steps for the remaining elements.

### 7.3 Written Exercises

1. Use the sorted list below and use binary search to look for Mike in the list. Show all the names that will be compared before Mike is found. Then, repeat the same process for Cathy (note: Cathy is not in the list).

   **Mike Betsy Aaron Steven Doug Pat Elise**

2. What is the benefit of using binary search over linear search?

3. (Warning: this question is difficult) The line:
   
   ```
   int mid = (min + max) / 2;
   ```

   in the code for binary search above has a bug, but is very specific. It works in the vast majority of cases, though. Figure out what the bug is, and what can be done to fix it (hint: think about the bounds on the values for int).

4. Write the contents of the following array after each step of the Selection Sort and Insertion Sort algorithms (assume the sorting is done in alphabetical order).

5. Why is Bubble Sort such an inefficient algorithm?

6. For both Insertion and Selection Sort, give an example of an input where the algorithm will finish as soon as possible, and another for as long as possible (Hint: what if the array is already sorted?)

   **Mike Betsy Aaron Steven Doug Pat Elise**
7.4 Programming Exercises

1. Implement Bubble Sort.
2. Implement Selection Sort.
3. Implement Insertion Sort.