

REVIEWING JAVA

SECOND EDITION

The Best Resource for Beginners

Alex Maureau

Visit:
www.cstutoringcenter.com/problems
for some fun programming challenges in Java.



CSTutoringCenter.com
Tutoring, challenges & more!

Also, visit **www.cstutoringcenter.com/reviewingjava** to download all the source code for the examples in each section. When prompted, enter the code **javaCodeLULU** (exactly as it appears, as it is case sensitive). You will then be granted access to the source files for each chapter's examples.

Copyright © 2009, 2011, 2013, 2019 by Alex Maureau.
Edited by Dani Maureau.
ISBN: 978-0-557-04355-2

All rights reserved, including the right of reproduction,
in whole or any part.

Special Thanks To

Department of Computer Science, CUNY Queens College

Without them, I would not be where I am today.

Dani Maureau

My beautiful, supportive wife & the English teacher who helped edit this book. I love you now and forever.

Matthew McClure

Computer Science wizard who initially inspired me to update my books with his total nerdgasm once he found out I wrote them.

Anne Smith-Thompson

For making last minute contributions and changes to this book. And for being a great professor and friend!

Stephanie Gallagher

For giving me the last bit of inspiration I needed to update this book. And who also gave me this funny exchange:

Her: "I saw him with your book and I was like 'I know that guy!'"

Me: "That's awesome. What a small world!"

Her: "And I thought, 'Oh, Java? Like Java the Hut!'"

Me: "It's not Java the Hut!!...but that's so going in the book now."

PREFACE

As many students embark on learning Java, some find it a challenge, and others find it as easy as others find it impossible. I've learned that there are many students who are willing to just give up at the drop of a hat or even after they fail miserably on the first exam (i.e. the " JOptionPane" exam).

During my own experiences, I realized that there are quite a few students who do not want to come into my office and ask me questions. To this day I still do not know why. Perhaps it is the stigma of the tutor or aide; too nerdy or "geeky" for some, or not important enough to others. I was once the latter, but quickly snapped out of it once I embarked on my computer science career.

This material is written to give hope to those lost students who have to take a programming class as it is mandated by their degree requirements, or those students who are anxious to learn a new subject. I hope you find this information helpful and understandable, so that you have a pleasant experience with Java programming. The material contained herein is a summary of the major topics of Java.

TABLE OF CONTENTS

Material

CHAPTER 1

<i>An Introduction to Java</i>	9
--------------------------------------	---

CHAPTER 2

<i>Variables & Operators</i>	16
--	----

CHAPTER 3

<i>Wrapper Classes & Parsing</i>	39
--	----

CHAPTER 4

<i>Input/Output Techniques</i>	49
--------------------------------------	----

CHAPTER 5

<i>Decisions & Logic</i>	59
------------------------------------	----

CHAPTER 6

<i>Strings in Java</i>	88
------------------------------	----

CHAPTER 7

<i>Looping</i>	114
----------------------	-----

CHAPTER 8

<i>Methods</i>	143
----------------------	-----

CHAPTER 9

<i>Recursion</i>	175
------------------------	-----

CHAPTER 10

<i>Arrays</i>	196
---------------------	-----

CHAPTER 11

<i>Some Useful Libraries</i>	249
------------------------------------	-----

CHAPTER 12

<i>Some Sorting Techniques</i>	278
--------------------------------------	-----

CHAPTER 13

<i>Working with Classes & Objects</i>	288
---	-----

CHAPTER 14

<i>Inheritance</i>	309
--------------------------	-----

CHAPTER 15

<i>Exceptions</i>	338
-------------------------	-----

CHAPTER 16

<i>File Input/Output</i>	351
--------------------------------	-----

CHAPTER 17

<i>Abstract Classes & Interfaces</i>	369
--	-----

CHAPTER 18

<i>Threading & Multitasking</i>	395
---	-----

CHAPTER 19

<i>Introduction to Generics, Collections, & Enums</i>	408
---	-----

CHAPTER 20

<i>Data Structures & the java.util Library</i>	430
--	-----

CHAPTER 21

<i>Dealing with Dates & Times</i>	482
---	-----

Appendices**APPENDIX A**

<i>ASCII Chart</i>	521
--------------------------	-----

APPENDIX B

<i>Number Conversions</i>	522
---------------------------------	-----

APPENDIX C

The Game of Keno 535

APPENDIX D

Programming Challenges 565

APPENDIX E

Jeopardy! Fun! How Much Can You Win on the Game of Jeopardy! 575

CHAPTER 1

An Introduction to Java

This chapter covers a basic introduction to the Java language, including the standard “Hello World” program.

TOPICS

1. <i>Console Input/Output</i>	10
2. <i>Hello World Program</i>	11
3. <i>Command Line Arguments</i>	12
4. <i>Documentation & Javadoc</i>	14
5. <i>Terminating Programs</i>	15

Java is a **high level programming language** that is strictly object-oriented. That means there is nothing but **objects** and **classes** that make up the language. So what parts are there to a class?

Constructor

A constructor will build the class. A class or object does not exist until you construct it. All classes are instantiated with the word **new**.

Instance variables (see Chapter 2)

There are data variables that are part of **each instance** of a class. These can be integers, Strings, characters, etc... or any other Object.

Methods (see Chapter 8)

These are the most essential part of a class, since these do work on a variable, Object, etc...

Much more on the above will be seen as this book progresses.

All Java programs, **AT THE BARE MINIMUM**, will look like this:

```
public class Name{  
    public static void main(String args[]){  
        //code here for program  
    }  
}
```

Where in the above, *Name* is the useful name of the object (class) for the start of the program; and the main method always looks like that. The argument of the main method is an array of Strings that represent command line arguments.

A lot has happened here in just a short time. Each part will become clearer as each chapter unfolds.

CONSOLE INPUT/OUTPUT

All basic input and output streams are contained in the System library. Below is a description of some methods contained in that library:

```
System.out.print( things to print )
```

This will simply print something to the console. It can print a variable, String, character, double, Object, etc... When attempting to print something, it will look for a `toString` method that will handle the conversion needed. This method does not print a new line character at the end.

```
System.out.println( things to print )
```

This will function the same as the above `print`, but it now will print the new line character at the end; hence the "ln" contained in the method name.

For console input, please see Scanners in Chapter 4.

"HELLO WORLD" PROGRAM

Let's create the most basic program. The program is collectively called the "Hello World" program, which will simply print a message to the console.

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

As stated, the above program will print the message to the console.

COMPILING JAVA PROGRAMS

To compile a Java program, you need to use the built in compiler for the command line called `javac`. That stands for "java compiler." The general format for compiling a program is:

```
javac FileName.java
```

where *FileName* is the name of the java file to compile. To compile multiple Java programs all at once, you may do this:

```
javac *.java
```

where the * will compile all filenames with the java extension. So using the above example of “Hello World,” here is how to compile it:

```
javac Hello.java
```

COMMAND LINE ARGUMENTS

A Java program can contain what’s called a command line argument. When compiling a program, you can give certain values to the main method’s array of arguments.

EXAMPLE 1: *Command Line Arguments*

Let’s see an example that will allow you to enter your first and last name in the command line.

```
public class Example1{
    public static void main(String args[]){
        if(args.length != 2){
            System.out.println("First + Last name needed");
            System.exit(1);
        }
        System.out.println("Hello " + args[0] + " "
                           + args[1]);
    }
}
```

The above program will execute first by checking the number of command line arguments (more details emerge on this aspect later in Chapter 10). It will then print out the name entered by the user, since we know the arguments are correct. The user’s first name is contained in args[0] and the last name is contained in args[1]. More on Strings and arrays will follow later.

Take note that in the above the + operator is used to join (or concatenate) Strings.

Here is the command line for compiling and executing. To compile the program:

```
javac Example1.java
```

To run the program, simply replace the message in the quotes with your first and last name, respectively, WITHOUT THE BRACKETS:

```
java Example1 [your first name] [your last name]
```

Also note that the arguments are separated by a space. This allows the array to note the separate arguments. Say that we ran the program above with the following command line:

```
java Example1 Alex
```

We will get an error message and the output would be:

First + Last name needed

We get the error message because the program did not contain the proper number of command line arguments. Now if we ran the program with this command line:

```
java Example1 Alex Maureau
```

We get the output:

Hello Alex Maureau

Please see Chapter 3 (Wrapper Classes & Parsing) for more on command line arguments and their use.

DOCUMENTATION & JAVADOC

An important part of programming is documentation. Without it, you can run into a lot of trouble when trying to debug a program. There are three types of documentation for a program in Java; **in-line comments**, **block comments** and **Javadoc comments**.

The first type of comment is an **in-line comment** that may look like this:

```
public static void main(String args[]) {  
    //check for arguments here  
}
```

The in-line comment is denoted by the two forward slashes. Anything appearing on that line past those two markers is ignored by the compiler.

The second type of comment is the **block comment** that may look like this in a code snippet:

```
public static void main(String args[]) {  
    /*  
     *check for arguments here  
     */  
}
```

Note that to START the block comment, there is the `/*` while to END the block comment, there is the `*/`. They are different from each other so be careful.

The final piece of commenting is **Javadoc**. The Javadoc comments are written thusly:

```
/**  
 *      Javadoc writing here. The Javadoc will explain  
 *      and document a method.  
 */  
public static void main(String args[]) {  
    //code here for main  
}
```

One important note is to realize that there is a `/**` to begin the Javadoc and the `*/` to finish it. If you are using a fancy schmancy environment such as Eclipse, it will turn blue to signify Javadoc. A block comment and in-line comment will turn green.

A good view of Javadoc is to take a look at the current Java APIs online at <https://docs.oracle.com/en/java/>

TERMINATING PROGRAMS

To terminate a program earlier than it should be, you can use the `exit()` method in the `System` library. The syntax will be as follows:

```
System.exit(1);
```

You can of course replace the number 1 with any integer of your choosing. Mostly, 0 and 1 are used. One is used to denote a problem and signal to a debugger that there is an early termination. Zero is used to state that there are no issues. Again, it is not a requirement to just use 0 and 1, but rather the most commonly-used integers for this purpose.

CHAPTER 2

Variables & Operators

This chapter discovers what variables are and how we can work with them in a program. Variables are essential to programming. This chapter covers the many different data types of variables and how we can operate on them.

TOPICS

1. Declaring Variables	17
2. Data Types	17
3. Variable Exercises	19
4. Operators	19
5. Operator Exercises	27
6. Example Programs	28
7. Types of Variables	31
8. Type Casting	34
9. Overflow	35
10.Overflow Exercises	37

In Java, a **variable** will be used to hold some form of data. That data can be of any type, such as an integer, decimal or character, or a custom data type (we will explore that later in the book), just to name a few.

While variables hold some form of data, we want to be able to work with them in a program, or “operate” on them as it’s called. By operating on variables, you can assign information/data to them, modify data, perform mathematical operations, perform logical operations -- basically anything you need to do run your program.

DECLARING VARIABLES

There are numerous ways to use and declare variables in Java. Here is one of the ways to declare a variable:

```
[identifier] type var_name;
```

Where in the above, *identifier* can be *public*, *private*, *protected* or nothing (see later); *type* is the data type of the variable and *var_name* is a useful name of the variable.

You can also declare a variable and assign a value to it. This is called **initializing** the variable.

```
[identifier] type var_name = initial_value;
```

Where in the above, *identifier* can be *public*, *private*, *protected* or nothing (see later); *type* is the data type of the variable; *var_name* is a useful name of the variable; and *initial_value* is the default value given to a variable.

DATA TYPES

While we now know how to declare a variable, we need to know both the different data types, as well as how to operate on them. The below chart consists of the **primitive** data types used in Java (note: String is not primitive, but will be used frequently throughout the book).

Type Name	Description	Size
char	Character or small integer. Signed range: -128 to 127 Unsigned range: 0 to 255	1 byte
short	Short Integer. Signed: -32768 to 32767 Unsigned: 0 to 65535	2 bytes
int	Integer. Signed: -2147483648 to 2147483647 Unsigned: 0 to 4294967295	4 bytes
long	Long Integer. Signed: -2147483648 to 2147483647 Unsigned: 0 to 4294967295	4 bytes
boolean	Boolean value. Either true or false. Acceptable as 1 or 0, respectively.	1 byte
float	Floating point number. 3.4e +/- 38 (7 digits)	4 bytes
double	Double precision floating point number. 1.7e +/- 308 (15 digits)	8 bytes
String	A class dedicated to strings in Java (more on this in Chapter 6). Note, the S is capitalized and must be in order for Java to recognize this type.	

Let's show a few examples of declaring variables below with the certain data types:

```
//declares an integer variable named x
int x;

//declares a double variable named avg
double avg;

//declares a Boolean variable named truth
boolean truth;

//declares a short variable named sh
short sh;

//declares a String variable named first_name
String first_name;
```

VARIABLE EXERCISES

DIRECTIONS: Answer each question below to the best of your ability. Solutions on page 588.

Problem 1:

Declare each of the following types of variables. Do not initialize, just declare them.

- a. A double precision variable called *num*.
- b. A float variable called *flt*.
- c. Two short variables called *s1* and *s2*.
- d. Four integer variables called *x1*, *x2*, *y1* and *y2*.
- e. A long int variable called *zzz*.
- f. Three char variables called *AA*, *BB* and *CC*.
- g. Two float variables called *F1* and *F2*.
- h. A boolean variable called *isPrime*.
- i. a String variable called *last_name*;
- j. A short variable called *shorty*.

Problem 2:

Declare and initialize each of the above variables in problem 1. You may choose to initialize them to *any value* (in its correct range), as well as any way you wish. They must match the data type of the variable.

OPERATORS

Now that we know a bit about variables, we can now know how to operate on them. An **operator** performs an action on a variable(s). It can assign values (the = operator), perform arithmetic (+, -, *, /), concatenate *String* objects (the + operator), as well as many more things that we will explore.

Below is a collection of operators and the category under which each falls. Examples for each of these operators will follow the chart.

Operator	Type	Name	Function
=	Assignment	Assignment	Assigns a value to a variable(s)
+	Arithmetic/ Concatenation	Addition	Performs the addition of a variable(s). Also can concatenate strings.
-	Arithmetic	Subtraction	Performs the subtraction of a variable(s)

*	Arithmetic	Multiplication	Performs the multiplication of a variable(s)
/	Arithmetic	Division	Performs the division of a variable(s)
%	Arithmetic	Modular Division	Performs the modular division of a variable(s)
++	Arithmetic	Pre/post increment	Increase the value of a variable by 1
--	Arithmetic	Pre/post decrement	Decrease the value of a variable by 1
==	Relational	Equals	Evaluates whether both sides of an expression have the same truth value.
!=	Relational	Does not equal	Evaluates whether both sides of an expression do not have the same truth value.
>	Relational	Strictly greater than	One side of an expression must be strictly greater than the other side in order to return an overall value of true.
>=	Relational	Greater than or equal to	One side of an expression can be greater than or equal to the other side, in order to return an overall value of true.
<	Relational	Strictly less than	One side of an expression must be strictly less than the other side in order to return an overall value of true.
<=	Relational	Less than or equal to	One side of an expression can be less than or equal to the other to return an overall value of true.
&&	Logical	Logical And	All elements of an expression must be true in order to return an overall value of true.
	Logical	Logical Or	One side of an expression must be true in order to return an overall value of true.
!	Logical	Logical Not (or Negate)	This will invert (or negate) a piece of or the entire expression, turning true to false and false to true.
+=	Compound Assignment/ Compound Concatenation	Plus Equals	Performs the addition of an expression or concatenation of String objects and then assigns that value to the variable.

--	Compound Assignment	Minus Equals	Performs the subtraction of an expression and then assigns that value to the variable.
*=	Compound Assignment	Multiply Equals	Performs the multiplication of an expression and then assigns that value to the variable.
/=	Compound Assignment	Divide Equals	Performs the division of an expression and then assigns that value to the variable.
%/	Compound Assignment	Mod Equals	Performs the modular division of an expression and then assigns that value to the variable.

Assignment Operators

= **Assignment**

This operator assigns a value to a variable(s).

Such examples can be:

```
//assigns the value 4 to the declared integer variable x
int x;
x = 4;

//declares a double variable named avg and initializes it to 0.0
double avg = 0.0;

//declares a Boolean variable named truth and assigns true to it
boolean truth = true;

//declare two String variables fname and lname and assign values
String fname, lname;
fname = "Alex";
lname = "Maureau";
```

This operator also can assign multiple values at once:

```
//declares two short variables x and y and assigns 5 to both of them
short x, y;
```

```
x = y = 5;  
  
//declares 3 char variables and assigns the character 'C' to all 3  
//notice that the only variable that is initialized is c  
char a, b, c = 'C';  
a = b = c;
```

Arithmetic Operators

+ Addition

Performs the addition of a variable(s).

- Subtraction

Performs the subtraction of a variable(s).

*** Multiplication**

Performs the multiplication of a variable(s).

/ Division

Performs the division of a variable(s).

Such examples can be:

```
//assume the following to begin:  
int sum = 0, diff = 0, prod = 0, quotient = 0, x, y;  
  
//finds the sum of the variables named x and y and assigns the value to  
//the sum variable  
sum = x+y;  
  
//finds the difference of the variables named x and y and assigns the  
//value to the diff variable  
diff = x-y;  
  
//multiplies the variables named x and y and assigns the value to the  
//prod variable  
prod = x*y;  
  
//divides two variables named x and y (ASSUMES Y IS NOT ZERO!) and  
//assigns the value to the quotient variable  
quotient = x/y;
```

There are also times when you may want to perform modular division. This can be represented as follows:

% Modular Division

The modulo division of two numbers is the remainder when the number on the left is divided by the number on the right.

Such examples can be:

3 % 5

Evaluates to 3 as 5 goes into 3 ZERO times, with a remainder of 3.

19 % 10

Evaluates to 9 as 10 goes into 19 ONE time, with a remainder of 9.

7 % 3

Evaluates to 1, since 3 goes into 7 TWO times, with a remainder of 1.

15 % 1

Evaluates to 0, since any number MOD 1 is 0 as 1 goes into that number fully.

12 % 0

This will evaluate to a compiler error, since you cannot divide by 0.

4 % 4

This will evaluate to 0, since any number MOD itself goes into itself fully with no remainder.

Now an actual code snippet showcasing modular division:

```
//declares two integer variables a and b  
int a, b, mod;  
  
//assign initial values  
a = 10;  
b = 5;  
  
//perform the modular division (value of mod would be 0 as 10%5 is 0)  
mod = a % b;
```

Relational/Equality Operators

== Logical Equals

Both sides of a statement must have the EXACT same value in order to return an overall value of true.

!= Logical Not Equals

Both sides of a statement must have DIFFERENT values in order to return an overall value of true.

> Logical Strictly Greater Than

A value on the left hand side of the operator must be strictly greater than the value on the right hand side of the operator in order to return an overall value of true.

< Logical Strictly Less Than

A value on the left hand side of the operator must be strictly less than the value on the right hand side of the operator in order to return an overall value of true.

>= Logical Greater Than or Equal To

A value on the left hand side of the operator can be greater than or equal to the value on the right hand side of the operator in order to return an overall value of true.

<= Logical Less Than or Equal To

A value on the left hand side of the operator can be less than or equal to the value on the right hand side of the operator in order to return an overall value of true.

Such examples can be:

(7 == 5)

Evaluates to FALSE as 7 does not equal the value of 5.

((4 + 5) == (5 + 3 + 1))

Evaluates to TRUE, since 9 is equal to 9. This performs the operations inside the parentheses first and then evaluates the logic.

(5 % 2 == 1)

Evaluates to TRUE, since 5 MOD 2 is 1. This first evaluates the modular division and then evaluates the logic.

(9 > 12)

Evaluates to FALSE, since 9 is not strictly greater than 12.

(9 >= 9)

Evaluates to TRUE, since 9 is greater than or equal to 9.

(12 - 4 <= 7 * 2)

Evaluates to TRUE, since 8 is less than or equal to 14.

(4 != (3 + 1))

Evaluates to FALSE as 4 is, in fact, equal to 4. This performs the action inside the parentheses first and then evaluates the expression.

((6 - 3 * 2) == (10 % 1))

Evaluates to FALSE, since 0 (value from the left parentheses) does not equal 1 (value from the right parentheses). Recall that anything MOD 1 will evaluate to 0.

Logical Operators

&& Logical And

ALL elements of the statement must be true in order to return an overall value of true.

|| Logical Or

One or more elements of the statement must be true in order to return an overall value of true.

! Logical Not

This will "invert" or "negate" a piece of the expression. In other words, if a piece of the expression evaluates to true, this will invert that to false and vice versa.

Some examples can be:

((4 != 3) && (3 != 4))

Evaluates to TRUE, as both sides of the expression evaluate to true. With the logical && operator, both sides need to have the same truth value, which in this case, they do.

((4 != 3) || (5 != 5))

Evaluates to TRUE, since one side of the expression evaluates to true. With the logical || operator, only one side needs to have a truth value of true in order for the entire statement to be true.

(((10 - 3) * 4) == 27 || ((14 * 2) != 18 && 4 < 5))

Evaluates to TRUE as the right side of the expression evaluates to TRUE, since 28 does not equal 18, and 4 is, in fact, strictly less than

5. The logical OR only requires one side of the expression to be true in order to return the overall value of true.

!(7 == 5)

Evaluates to TRUE as 7 does not equal the value of 5 however, we are inverting the overall value by use of the ! operator.

Increase/Decrease Operators

++ Pre/Post Increment

Increase the value of a variable by 1. If the ++ comes before the variable you want to increment, it will increase the value FIRST before using it. If it comes AFTER the variable, the program uses the current value of the variable first, THEN increases it.

++k -> Increment, THEN use
k++ -> Use, THEN increment

-- Pre/Post Decrement

Decrease the value of a variable by 1. If the -- comes before the variable you want to decrement, it will decrease the value FIRST before using it. If it comes AFTER the variable, the program uses the current value of the variable first, THEN decreases it.

--k -> Decrement, THEN use
k-- -> Use, THEN decrement

Compound Assignment Operators

+= Plus Equals

Shorthand operator for adding and then assigning a value to a variable. This can be translated to something similar to the below:

a += 3 MEANS a = a + 3;
x += x MEANS x = x + x;

-= Minus Equals

Shorthand operator for subtracting and then assigning a value to a variable. This can be translated to something similar to the below:

a -= 3 MEANS a = a - 3;
x -= x MEANS x = x - x;

***= Multiply Equals**

Shorthand operator for multiplying and then assigning a value to a variable. This can be translated to something similar to the below:

<code>a *= 3</code>	MEANS	<code>a = a * 3;</code>
<code>x *= x</code>	MEANS	<code>x = x * x;</code>

/= Divide Equals

Shorthand operator for dividing and then assigning a value to a variable. This can be translated to something similar to the below:

<code>a /= 3</code>	MEANS	<code>a = a / 3;</code>
<code>x /= 8</code>	MEANS	<code>x = x / 8;</code>

%= Mod Equals

Shorthand operator for performing modular division and then assigning a value to a variable. This can be translated to something similar to the below:

<code>a %= 3</code>	MEANS	<code>a = a % 3;</code>
<code>x %= 2</code>	MEANS	<code>x = x % 2;</code>

Each operator above may have different meanings when dealing with different topics. For example, the + operator relates to addition, but it can also concatenate strings. More on those scenarios as the book progresses.

OPERATOR EXERCISES

DIRECTIONS: Answer each question below to the best of your ability. Solutions on page 588.

Problem 1:

Identify each of the following operators and briefly explain what it does:

- a. &&
- b. ||
- c. =
- d. +=
- e. %
- f. ++
- g. %=
- h. *
- i. <
- j. >=
- k. ==

Problem 2:

Given the following examples, answer to the best of your ability:

- a. $12 \% 3$ _____
- b. $4 \% 7$ _____
- c. $6 \% 3$ _____
- d. $77 \% 10$ _____
- e. $100 \% 100$ _____
- f. $99 \% 1$ _____
- g. $(9 \% 9) \% 9$ _____
- h. $((44 \% 10) \% 10) \% 10$ _____

Problem 3:

Given the following snippets, what will be produced as the truth value:

- a. $(9 \leq 16)$ _____
- b. $(12 > 12)$ _____
- c. $!(12 > 12)$ _____
- d. $!(!!(4 == 4))$ _____
- e. $((55 > 99) \mid\mid (44 < 99))$ _____
- f. $(3 != 4 \&\& 3 != 5)$ _____
- g. $!(5 != 5 \&\& 4 \leq 5)$ _____
- h. $(!(!(!!(8 < 8))))$ _____

EXAMPLE PROGRAMS

Below are some examples that show the use of most of the above operators. Each example is explained in detail following the code. *NOTE: Some of these programs will be modified later in the book when dealing with wrapper classes and objects.*

EXAMPLE 1: Arithmetic Showcase

This program will perform addition, subtraction, multiplication, division, and modular division on two integers.

```
public class Example1{  
    public static void main(String args[]){  
        //declare two variables  
        int a = 5, b = 4;
```

```

    //addition
    System.out.println("The sum is: " + (a+b));

    //subtraction
    System.out.println("The difference is: " + (a-b));

    //multiplication
    System.out.println("The product is: " + (a*b));

    //division
    System.out.println("The quotient is: " + (a/b));

    //modular division
    System.out.println("The a mod b is: " + (a%b));
} //main
} //class

```

A sample run of the program will produce the following output:

```

The sum is: 9
The difference is: 1
The product is: 20
The quotient is: 1
The a mod b is: 1

```

EXAMPLE 2: *Change in Coins*

This program will calculate the change needed in the number of quarters, dimes, nickels and pennies.

```

public class Example2{
    public static void main(String args[]){
        int x = 90;

        //print message to user
        System.out.println("Change for " + x + " cents:");

        //quarters, dimes, nickels and cents wanted
        int q, d, n, c;

        q = x / 25;
        x = x % 25;
        d = x / 10;
    }
}

```

```

x = x % 10;
n = x / 5;
c = x % 5;

System.out.println("Quarters: " + q + "\nDimes: " + d +
    "\nNickels: " + n + "\nCents: " + c);
} //main
} //class

```

Say that the user needs 90 cents in change. The first thing that the program needs to do is calculate the quarters. Here, 90 divided by 25 is 3. Note that the int DOES NOT handle decimals and only looks at whole numbers. It then stores 3 into the variable q.

Then, the new value of x is calculated by placing the remainder when 90 is divided by 25. Here, it is 15.

Now, the dimes are calculated by taking 15 (since that is current value of x) and dividing it by 10 which produces 1 (again, the int ignores the decimals). Now, the new value of x is 15 mod 10, which is 5.

The program will then calculate the nickels and cents accordingly.

The output from running the above program is:

```

Change for 90 cents:
Quarters: 3
Dimes: 1
Nickels: 1
Pennies: 0

```

EXAMPLE 3: Incrementing/Decrementing

This program will show the use of the operators for incrementing and decrementing a number. This also shows the order of the operators.

```

public class Example3{
    public static void main(String args[]){
        int num = 5;

        //allows user to enter an integer
    }
}

```

```
System.out.println("The number 5:");

//once entered, use the operators
//"post" increment:
System.out.print(num++);

//"pre" increment:
System.out.print(++num);

//"post" decrement:
System.out.print(num--);

//"pre" decrement:
System.out.print(--num);
} //main
} //class
```

Let's see why the output is this way. First, the value of the variable num is 5. The first operator that is used on the variable is the "post" increment operator. This uses the variable first (for output) THEN increments it. So the output is 5, BUT THE VARIABLE HOLDS THE VALUE 6.

Next, the variable is "pre" incremented so it is incremented FIRST, THEN outputted. The output is 7, since it was already incremented. The same applies for the decrementing.

The output from running the above program is:

The number 5:
5775

TYPES OF VARIABLES

While we have a basic understanding of variables and operators, we need to know that there are different categories of variables used in this language. Those are **local variables**, **static variables**, and **instance variables**. But first, we must learn about access levels.

Access Levels

As we observed earlier in the chapter when declaring a variable:

```
[identifier] type var_name;
```

Where *identifier* can be either public, private or protected, or nothing; *type* is the data type of the variable; and *var_name* is a useful name of the variable.

Access levels determine whether or not other classes can see and use a particular method or variable. There are 4 types of access levels as can be declared:

Public

A class, variable, or method can be declared public, in which case it is visible to all classes everywhere.

Private

A variable or method can be declared private, but they cannot be accessed outside the class in which they appear. A class cannot be declared private.

Default [no access level declared]

A default level of access is declared by not writing any identifier before a class, variable, or method. The class, variable, or method is only visible within its package.

Protected

Methods or data members declared as protected are accessible within the same package or subclasses in a different package.

Local Variables

Local variables are just as they sound; they are only able to be accessed & used within the method or block of code in which they are declared. For example, see the program below:

```
public class Example{
    public static void main(String args[]) {
        int x = 4;
        char c = 'C';
        System.out.println(x);
    }
}
```

The int variable *x* and the char variable *c* are local to the main() method in the Example class. They can not be accessed or used anywhere else in the program. The output from above will simply be 4, since that is the variable's initial value.

Instance Variables

A class or object can have its own variables. These are what's called **instance variables**. By its definition, an instance variable is unique to each instance of the object; so in general, each time a class is constructed, there is another instance of the variable associated with it.

These variables are declared outside any methods you may have, including main(). These variables are declared either public, private, protected, or with no identifier, as mentioned earlier. They are global to the class or object in which they appear. Let's see a small example.

```
public class Variables{
    //declare instance variables
    private int x = 6;
    public double d = 0.0;
    private char c = 'A';

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

Static Variables

Static variables are quite a bit harder to understand. Do not be confused with a static variable from C++. They are very different.

In Java, a static variable (also called a **class variable**), is a variable that is given a fixed block of memory. The static keyword tells the compiler that there is **exactly one** copy of this variable in existence, no matter how many times the class has been constructed.

Let's just think of a real world example. Say you own a car. The car will always have four wheels. If you made a class in Java called *Car*, a variable called *numWheels* can be

made static, since it will be the same for every car. Similarly with fruit, an orange will always be an orange color, so a *color* variable in an Orange class can be static.

Final Variables

In C++, we can declare variables constant if we know their values will not change. In Java, we do not use the keyword **const**, but we use the keyword **final**. Final variables cannot be changed and are made constant. They can be either instance, static or local variables or even local to a method.

A common practice is to make the final variable's name all capital letters. So here is what something may look like:

```
public class Car{  
    static final int WHEELS = 4;  
  
    //rest of code below  
}
```

TYPE CASTING

In Java, we can allow a variable to change its current data type by casting it to another data type. There are two types of casting: **implicit** and **explicit**.

Implicit Casting

Implicit casting is when the data type is changed automatically in the program. We do not need to explicitly tell the program to cast.

Automatic casting

This occurs in the compiler as it tries to compile your program:

```
int x = 5.667; //automatically changed to 5  
float f = 7; //automatically changed to 7.0
```

“Promotion” casting

This occurs when more space is given to a number (i.e. short to int or int to long):

```
int I = 6;  
long u = I; //promoted here to long.
```

"Demotion" casting

This occurs when less space is given to a number (i.e. int to short):

```
double d = 8.9;
float f = d;    //demoted here from double.
```

Explicit Casting

Explicit casting is when we need to tell the program to change the type. This is done by placing the data type in parentheses next to the expression.

Say we wanted to compute the average of some integers, and just wanted to output the whole number from the average, ignoring the decimal places. Here is a program that will do this:

```
public class Example{
    public static void main(String args[]){
        int a = 5, b = 7, c = 10;
        double avg = ((double) a+b+c) / 3.0;  //cast here

        int ans = (int) avg;  //cast here
        System.out.println(ans);
    }
}
```

We cast in two places with the above program. First, we cast the sum of the 3 integers to a double precision value and divide by 3.0. We need to do this, otherwise we get incorrect division.

We also cast in the final result changing the answer from the double precision value to an int value. This program will output 7 when run, but note that the real average is 7.33333333333333.

OVERFLOW

In Java, variables can overflow their capacity. For instance, if you were to declare the following piece of code:

```
short z = 10000;
z = z + 40000;
```

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

You would see something like this as the output:

-15536

The reason behind this is that you ran out of room to represent the number in its binary form. That is a bit technical to discuss here, but just know that if you see a negative number when it is supposed to be positive, you overflowed the number!!! The solution is to use a bigger data type.

EXAMPLE 4: *Overflowed on Purpose*

Here is a program that will purposefully overflow some numbers.

```
public class Example4{
    public static void main(String args[]){
        System.out.println( (short)(10000) );
        System.out.println( (short)(33000) );
        System.out.println( (int)(10000000) );
        System.out.println( (int)(123456789) );
        System.out.println( (long)(1010101010101010) );
        System.out.println( (long)(9999999999999999) );
    } //main
} //class
```

The output from running the above program is:

10000
-32536
10000000
123456789
-714149102
1569325055

NOTE: Depending on the compiler you have, it may not even compile (for instance in Eclipse). If you use command line Java, it may work out to the output above.

The first number will work just fine (being the short(10000)), since that is in range. The following short(33000) is just a bit out of range, so it will give you the -32536 you see on screen.

Then, both the int values are still in range, so they will be just fine, however both the long values are not! It may appear that a long holds more numbers, but certainly not numbers that big!!!

CASTING EXERCISES

DIRECTIONS: Answer each question below to the best of your ability. Solutions on page 589.

Problem 1:

Observe each type of casting and explain what will be printed on screen with the `println` statement:

a.

```
int x = 99;
short xx = x;
System.out.println( xx );
```

b.

```
double c = 55.344;
System.out.println( ((int) c) );
```

c.

```
long n = 9999.999;
double nn = n;
System.out.println( nn );
```

d.

```
int q = 499;
double x = 2.1;
long qq = q + ((int) x);
System.out.println( qq );
```

Problem 2:

Identify which of the following values are out of range (or in other words, overflow) for the following lines of `println` statements:

- a. `System.out.println((int) 4999999);`
- b. `System.out.println((short) 32333);`
- c. `System.out.println((long) 444444444);`

- d. System.out.println((long) 444444444444);
- e. System.out.println((char) 257);
- f. System.out.println((double) 999999.9999999);
- g. System.out.println((float) 43*100+7);
- h. System.out.println((double) 777*777*10);
- i. System.out.println((short) -40000);
- j. System.out.println((int) -99955533322);

CHAPTER 3

Wrapper Classes & Parsing

This chapter covers how to convert Strings to numerical values. It also covers how to use this concept from the command line. There are some example programs that showcase the use of many different types of parsing.

TOPICS

1. <i>Wrapper Classes</i>	40
2. <i>Character Wrapper Class</i>	44
3. <i>Defining Numbers</i>	44

Each primitive data type in Java (int, short, long, float, double, char, and boolean) has its own wrapper class. But what is a wrapper class?

WRAPPER CLASSES

A **wrapper class** is simply an object version of a primitive data type. It contains methods or constants that can be useful for a program. Here is a list of the wrapper classes for the primitive data types:

Integer

*Methods for an **int** data type.*

Short

*Methods for a **short** data type.*

Long

*Methods for a **long** data type.*

Double

*Methods for a **double** data type.*

Float

*Methods for a **float** data type.*

Character

*Methods for a **char** data type.*

Boolean

*Methods for a **boolean** data type.*

Each of the above wrapper classes contains a very important method that will **parse** (or convert) any String, either a command line argument or a String object, into the appropriate data type. Those methods are defined below:

Integer.parseInt(String val)

*Returns an **int** value from a String representation of a number. A NumberFormatException is thrown if the argument is not a String representation of a number.*

Short.parseShort(String val)

*Returns a **short** value from a String representation of a number. A NumberFormatException is thrown if the argument is not a String representation of a number.*

Long.parseLong(String val)

Returns a long value from a String representation of a number. A NumberFormatException is thrown if the argument is not a String representation of a number.

Double.parseDouble(String val)

Returns a double precision value from a String representation of a number. A NumberFormatException is thrown if the argument is not a String representation of a number.

Float.parseFloat(String val)

Returns a float value from a String representation of a number. A NumberFormatException is thrown if the argument is not a String representation of a number.

(Exceptions will be explained further in Chapter 15).

A few examples will show some of the above in play.

EXAMPLE 1: *Modified Arithmetic Showcase*

This program will perform addition, subtraction, multiplication, division, and modular division of two integers. This is the modified version of the program from the previous chapter. This one uses two command line arguments and parses them accordingly to perform the arithmetic.

```
public class Example1{
    public static void main(String args[]){
        int a = 0, b = 0;

        //convert (parse) the entered Strings to an int value
        //with the use of the Integer wrapper class
        a = Integer.parseInt(args[0]);
        b = Integer.parseInt(args[1]);

        //addition
        System.out.println("The sum is: " + (a+b));

        //subtraction
        System.out.println("The difference is: " + (a-b));

        //multiplication
        System.out.println("The product is: " + (a*b));
```

```

    //division
    System.out.println("The quotient is: " + (a/b));

    //modular division
    System.out.println("The a mod b is: " + (a%b));

} //main
} //class

```

The difference between this program and the previous chapter is the fact that you are now using command line arguments.

A sample run of the program will produce the following output (with command line arguments 5 4:

```

The sum is: 9
The difference is: 1
The product is: 20
The quotient is: 1
The a mod b is: 1

```

EXAMPLE 2: *Modified Change in Coins*

This program will calculate the change needed in the number of quarters, dimes, nickels, and pennies. The value is entered from the command line and will be parsed to an int.

```

public class Example2{
    public static void main(String args[]){
        //convert the command line argument here with the use
        //of the Integer wrapper class
        int x = Integer.parseInt(args[0]);

        System.out.println("Change for " + x + " cents:");
        int q, d, n, c; //quarters, dimes, nickels and cents wanted

        q = x / 25;
        x = x % 25;
        d = x / 10;
        x = x % 10;
        n = x / 5;
        c = x % 5;
    }
}

```

```

        System.out.println("Quarters: " + q + "\nDimes: " + d +
                           "\nNickels: " + n + "\nCents: " + c);
    } //main
} //class

```

Running it off the command line and giving it the command line argument of 90, a sample run of this program from the above is:

Change for 90 cents:

Quarters: 3

Dimes: 1

Nickels: 1

Pennies: 0

EXAMPLE 3: *Finding the Average*

This program will take 3 command line arguments and calculate the average of those values. It makes use of the Double wrapper class to do the calculations as needed.

```

public class Example3{
    public static void main(String args[]){
        //the 3 numbers desired
        double n1, n2, n3, avg;

        //convert each argument entered to a double
        //value with the use of the Double wrapper class
        n1 = Double.parseDouble(args[0]);
        n2 = Double.parseDouble(args[1]);
        n3 = Double.parseDouble(args[2]);

        //find the average & display result
        avg = (n1 + n2 + n3) / 3.0;

        System.out.println("The average is: " + avg);
    } //main
} //class

```

The program output with command line arguments 5 10 12 will be:

The average is 9.0

CHARACTER WRAPPER CLASS

An important class that contains some useful methods is the character wrapper class. There are methods that will check if a character is a digit, letter, etc... as well as uppercase, lowercase, etc...

```
boolean isDigit(char c)
```

This method will return true if the character argument is a digit from 0 to 9. Else, it will return false.

```
boolean isLetter(char c)
```

This method will return true if the character argument is a letter from, EITHER UPPERCASE OR LOWERCASE. Else, it will return false.

```
boolean isUpperCase(char c)
```

This method will return true if the character argument is an uppercase letter. Else, it will return false.

```
boolean isLowerCase(char c)
```

This method will return true if the character argument is a lowercase letter. Else, it will return false.

```
boolean isWhiteSpace(char c)
```

This method will return true if the character argument is a white space character. Else, it will return false.

As seen in the program, these methods are utilized to keep track of the different types of characters in the String argument.

These will be shown in action in Chapter 6, when discussing Strings.

DEFINING NUMBERS

Each respective numeric wrapper class (Short, Integer, Long, Float, and Double), have methods that return its value as a primitive data type (short, int, long, float, and double). They are defined below and can be accessed from the respective wrapper class:

```
short shortValue()
```

This method will return a short value representing the numeric value of the object.

```
int intValue()
```

This method will return an int value representing the numeric value of the object.

```
long longValue()
```

This method will return a long value representing the numeric value of the object.

```
float floatValue()
```

This method will return a float value representing the numeric value of the object.

```
double doubleValue()
```

This method will return a double precision value representing the numeric value of the object.

So far, we have just used primitive data types for our desired numeric variables.

However, it is perfectly acceptable to do something like this:

```
Integer num = new Integer("10");
Short num = new Short("100");
Double num = new Double("5.54321");

Etc...
```

It is also acceptable to do the following (good practice is to be sure the values will not overflow):

```
Integer num = 10;
Short num = 100;
Double num = 5.54321;

Etc...
```

EXAMPLE 4: Defining Numbers

The below program will showcase the numerous value methods as described above.

```
public class Example4{
    public static void main(String args[]) {
```

```

Double n1 = new Double("4.59234");
Integer n2 = 5;

//display results for n1
System.out.println("current value: " + n1);
System.out.println("short value: " + n1.shortValue());
System.out.println("int value: " + n1.intValue());
System.out.println("long value: " + n1.longValue());
System.out.println("float value: " + n1.floatValue());
System.out.println("double value: " + n1.doubleValue());
System.out.println();

//display results for n2
System.out.println("current value: " + n2);
System.out.println("short value: " + n2.shortValue());
System.out.println("int value: " + n2.intValue());
System.out.println("long value: " + n2.longValue());
System.out.println("float value: " + n2.floatValue());
System.out.println("double value: " + n2.doubleValue());
} //main
} //class

```

The output from running the above program is:

```

current value: 4.59234
short value: 4
int value: 4
long value: 4
float value: 4.59234
double value: 4.59234

```

```

current value: 5
short value: 5
int value: 5
long value: 5
float value: 5.0
double value: 5.0

```

EXAMPLE 5: More Change in Coins

This program will again calculate the change needed in the number of quarters, dimes, nickels, and pennies. The value is entered from the command line.

```

public class Example5{
    public static void main(String args[]){
        //convert the command line argument here
        Integer x = new Integer(args[0]);

        System.out.println("Change for " + x + " cents:");
        int q, d, n, c; //quarters, dimes, nickels and cents wanted

        q = x / 25;
        x = x % 25;
        d = x / 10;
        x = x % 10;
        n = x / 5;
        c = x % 5;

        System.out.println("Quarters: " + q + "\nDimes: " + d +
                           "\nNickels: " + n + "\nCents: " + c);
    } //main
} //class

```

Running it off the command line and giving it the command line argument of 90, a sample run of this program from the above is:

```

Change for 90 cents:
Quarters: 3
Dimes: 1
Nickels: 1
Pennies: 0

```

EXAMPLE 6: Another Way to Find the Average

This program will take 3 command line arguments and calculate the average of those values. It makes use of the Double wrapper class to do the calculations as needed.

```

public class Example6{
    public static void main(String args[]){
        //the 3 numbers desired
        Double n1, n2, n3, avg;

        //convert each argument entered to a double
        //value with the use of the Double wrapper class
        n1 = new Double(args[0]);
        n2 = new Double(args[1]);
    }
}

```

```
n3 = new Double(args[2]);  
  
    //find the average  
    avg = (n1.doubleValue() + n2.doubleValue() +  
           n3.doubleValue()) / 3.0;  
  
    //display result  
    System.out.println("The average is: " + avg);  
} //main  
} //class
```

The program output with command line arguments 5 10 12 will be:

The average is 9.0

The output is the same as example 3 above. This was just another way of writing the code.

CHAPTER 4

Input/Output Techniques

This chapter discusses the different methods of input and output in Java. We will explore Scanners that handle input from the console and JOptionPanees that handle both input and output, but in a more graphical approach.

TOPICS

- | | |
|-------------------------|----|
| 1. <i>Scanners</i> | 50 |
| 2. <i>JOptionPanees</i> | 55 |

As mentioned in the first chapter, there are some different types of input and output techniques. There is the console technique using a **Scanner** object, and the graphical technique using a **JOptionPane**. When deciding on which one to use, it will depend on the type of Operating System you are using, since a JOptionPane will not work on a UNIX/LINUX environment.

SCANNERS

A Scanner will allow you to input data from the console at some point(s) in a program. A Scanner is part of the **java.util** library, so this needs to be imported in order to be used.

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

To define and use a Scanner in a program, most generally:

```
Scanner var_name = new Scanner(System.in);
```

Where in the above, ***var_name*** is a useful name for the Scanner object.

The Scanner class has some methods associated with it to gather input from a user:

```
nextInt()
Reads an int value from the user.

nextShort()
Reads a short value from the user.

nextLong()
Reads a long value from the user.

nextDouble()
Reads a double value from the user.

nextFloat()
Reads a float value from the user.

nextBoolean()
Reads a boolean value from the user.
```

nextLine()

Reads a **String** value from the user until a new line is reached.

next()

Reads a **String** value from the user until a whitespace is reached.

As we progress in this section, we will see examples of each.

EXAMPLE 1: Name and Age with Scanners

Here is a sample program that uses a Scanner to collect the first name and age of a user:

```
import java.util.Scanner;
public class Example1{
    public static void main(String[] args){
        String name = "";
        int age = 0;

        //setup the Scanner
        Scanner input = new Scanner(System.in);

        //get first name
        System.out.print("Please enter your first name: ");
        name = input.nextLine();

        //get age from user
        System.out.print("Please enter your age: ");
        age = input.nextInt();

        //display results
        System.out.println("Thanks " + name + "! You are "
            + age + " years old!");
    } //main
} //class
```

The program will prompt the user for the first name. It will stop and wait for as long as it takes for the user to enter their name. Once the user enters the name, the *name* variable will get the value entered by the user with the use of the nextLine() method.

The program continues & prompts for the user's age. Again, the program waits for the user to enter the data. Once complete, there is a simple message printed to the console.

A possible output could be:

Thanks Alex! You are 32 years old!

EXAMPLE 2: *Scanner Averages*

The program below will calculate the average of 3 numbers entered by the user from the console, and display the result. It makes use of the nextDouble() method of the Scanner class.

```
import java.util.Scanner;
public class Example2{
    public static void main(String args[]){
        //setup the Scanner
        Scanner s = new Scanner(System.in);

        //variables needed
        double n1, n2, n3, avg;

        //prompt user for the input
        System.out.println("Enter 3 numbers: ");
        n1 = s.nextDouble();
        n2 = s.nextDouble();
        n3 = s.nextDouble();

        //find average
        avg = (n1 + n2 + n3) / 3.0;

        System.out.println("The average is: " + avg);
    } //main
} //class
```

Running the above program will produce the following sample output:

Enter 3 numbers:

45

55

65

The average is: 55.0

You can also choose to enter the 3 numbers at once on the console, as in the below:

Enter 3 numbers: 45 55 65

The average is: 55.0

EXAMPLE 3: *Scanner Arithmetic Showcase*

The program below will gather two long type numbers from the user, and perform various arithmetic functions on them.

```
import java.util.Scanner;
public class Example3{
    public static void main(String args[]){
        //setup the Scanner
        Scanner s = new Scanner(System.in);

        //variables needed for input
        long n1, n2;

        //prompt user for the input
        System.out.println("Enter 2 numbers: ");
        n1 = s.nextLong();
        n2 = s.nextLong();

        //variables for the various needs
        double avg = 0.0;
        long sum, diff, prod, quot, mod;

        //initialize all variables to 0
        sum = diff = prod = quot = mod = 0;

        //find average
        avg = (n1 + n2) / 2.0;

        //find sum
        sum = n1+n2;
        diff = n1-n2;
        prod = n1*n2;
        quot = n1/n2;
        mod = n1 % n2;

        //display results
        System.out.println("The average is: " + avg);
        System.out.println("The sum is: " + sum);
        System.out.println("The difference is: " + diff);
        System.out.println("The product is: " + prod);
        System.out.println("The quotient is: " + quot);
        System.out.println("The remainder is: " + mod);
```

```
    } //main
} //class
```

Running the above program will produce the following sample output:

```
Enter 2 numbers: 50 100
The average is: 75.0
The sum is: 150
The difference is: -50
The product is: 5000
The quotient is: 0
The remainder is: 50
```

Or using even large numbers, since they are of the long type:

```
Enter 2 numbers: 400000 500000
The average is: 450000.0
The sum is: 900000
The difference is: -100000
The product is: 2000000000000
The quotient is: 0
The remainder is: 400000
```

EXAMPLE 4: Words & Sentences

The program below will gather a sentence from the user. It will then display the first two words of the sentence, followed by the rest of it with the use of next() and nextLine() from the Scanner class.

```
import java.util.Scanner;
public class Example4{
    public static void main(String args[]){
        //setup the Scanner
        Scanner s = new Scanner(System.in);

        //prompt user
        System.out.println("Enter a sentence: ");

        //declare variables for the first two words
        String word1 = "", word2 = ";
```

```

word1 = s.next(); //get first word
word2 = s.next(); //get second word

//get the rest of the sentence entered
String sentence = "";
sentence = s.nextLine();

//display output
System.out.println("Word 1: " + word1);
System.out.println("Word 2: " + word2);
System.out.println("Sentence: " + sentence);

} //main
} //class

```

Running the above program will produce the following sample output:

```

Enter a sentence: My car broke down this morning on the way to work
Word 1: My
Word 2: car
Sentence: broke down this morning on the way to work

```

Recall that next() will gather input from the user until a whitespace is reached, whereas nextLine() will gather input from the user until an end line is reached (this is denoted by a '\n' or newline character; the same for a carriage return or '\r' character). Strings will be discussed in much more detail in Chapter 6.

JOPTIONPANES

A JOptionPane is a graphical way of asking for data from a user. It is possible to write a program that only uses JOptionPanees for its input/output. A JOptionPane is part of the **javax.swing** library and needs to be imported into a program like so.

```
import javax.swing.JOptionPane;
```

There are two main methods to a JOptionPane as seen below:

```
String showInputDialog(Object message)
```

This method will take, as its argument, a String representing a message you are displaying to the user. It will return a String to the variable to which it is being assigned. Note that you will have to use the various wrapper classes to parse information as needed.

```
void showMessageDialog(Component c1, Object message)
```

This method takes two arguments. Here, the Component feature will always be null, since it won't be needed for basic examples. The message argument will be the message you want to display to the user. The return type of the method is void, since it is just going to be displaying information.

Some basic examples of JOptionPanees can be seen below:

EXAMPLE 5: JOptionPane Name and Age

This is the modified version of the previous example of asking the user for his or her name and age.

```
import javax.swing.JOptionPane;
public class Example5{
    public static void main(String[] args){
        String name = "";
        int age = 0;

        //get name from user with a JOptionPane
        name = JOptionPane.showInputDialog(
            null, "Please enter your name: ");

        //get age from user with a JOptionPane
        //need to parse as an int type
        age = Integer.parseInt(JOptionPane.showInputDialog(null,
"Please enter your age: "));

        //display message with a JOptionPane
        JOptionPane.showMessageDialog(null,
            "Hello " + name + ". You are " + age + " years old!");
    } //main
} //class
```

Messy code perhaps, however, it is not too bad after all. Firstly, the program will prompt the user for the name first by showing a JOptionPane input dialog. The

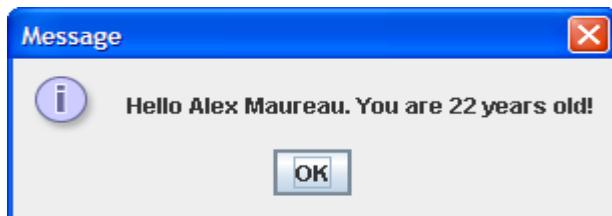
program will pause and wait for the user to enter the data. That input box looks like this one below:



Once the name is entered by the user, it will then show another box for the user to enter the age. It needs to parse the number to an integer, as all data entered in a JOptionPane are stored as Strings. That box looks like this below:



Finally, a message box is shown displaying the data that was entered by the user. A sample of what that box would look like is below:



EXAMPLE 6: *JOptionPane Averages*

Another example with a JOptionPane will take 3 numbers from the user, and output the average of them to a JOptionPane. The windows will not be displayed here, since the only difference is the messages in them:

```
import javax.swing.JOptionPane;
public class Example6{
    public static void main(String[] args) {
```

```
double num1, num2, num3;
double avg = 0.0;

//get first number and parse as a double
num1 = Double.parseDouble(
    JOptionPane.showInputDialog("Please enter 1st number:"));

//get second number and parse as a double
num2 = Double.parseDouble(
    JOptionPane.showInputDialog("Please enter 2nd number:"));

//get last number and parse as a double
num3 = Double.parseDouble(
    JOptionPane.showInputDialog("Please enter 3rd number:"));

//find the average
avg = (num1+num2+num3)/3.0;

JOptionPane.showMessageDialog(null, "The average is: " + avg);
} //main
} //class
```

CHAPTER 5

Decisions & Logic

This chapter covers basic decision making when programming, with the use of if statements, while statements and switch/case statements. This chapter also covers the syntax of decision making.

TOPICS

1. <i>If Statements</i>	60
2. <i>Nested If Statements</i>	64
3. <i>Empty Decisions</i>	67
4. <i>While Statements</i>	68
5. <i>Do/While Statements</i>	72
6. <i>Break & Continue</i>	74
7. <i>Switch/Case Statements</i>	76
8. <i>Exercises</i>	81

In all of programming, decisions need to be made. Most can be made with what is called an **if statement**.

Whether you realize it or not, we use if statements in our daily lives every day.

- If it is raining outside, then we will stay inside and watch a movie.
- If the fuel tank is low in the car, then we'll stop and get gas.
- If I'm not feeling well, then I will stay home from work. Etc...

IF STATEMENTS

Below is what an if statement looks like in Java:

```
if( condition(s) ){
    //code for the if
}
```

Where *condition(s)* is a logical true/false statement.

There may also be an *if / else if* statement. This kind of structure will check for a condition initially and *if and only if* that initial condition is FALSE, the else if clause is checked. If the *else if* clause is FALSE and there are no other statements to check, nothing will happen. There is no limit to how many *else if* clauses you may have.

```
if( condition(s) ){
    //code for the if
} else if( condition(s) ){
    //code for the else if
}
.
.
.
} else if( condition(s) ){
    //code for the else if
}
```

To keep a program's logic short and sweet, a more simplistic way of doing things is to make an *if / else* block. Here, the *else* clause is a general condition and will handle actions when the initial *if or else if* is FALSE. Here is what that would look like:

```

if( condition(s) ){
    //code for the if
}else if( condition(s) ){
    //code for the else if
}else{
    //code for the else
}

```

Notice that there is only 1 else clause and that it does not contain any conditions. In short, the rule about if statements is that the condition(s) inside the if **MUST BE TRUE** in order for the if statement to work (or be "tripped" as some say). If it is FALSE, the else if or else statements are checked.

EXAMPLE 1: Even / Odd Detection

The objective of the below program is to detect whether or not a number is odd or even. This requires a test of modulo division. The definition of an even number is when its remainder when divided by 2 is 0.

```

// checks for an even or odd number
public class Example1{
    public static void main(String args[]){
        int num = 5;

        //determine if number is even or odd
        if(num % 2 == 0){
            System.out.println(num + " is even!");
        }else{
            System.out.println(num + " is odd!");
        }

        //now try it again with another value
        num = 4;

        //determine if number is even or odd
        if(num % 2 == 0){
            System.out.println(num + " is even!");
        }else{
            System.out.println(num + " is odd!");
        }
    } //main
} //class

```

The output of the above program will be:

```
5 is odd!  
4 is even!
```

Why? Because the first time the number is checked with the if statements shows that 5 MOD 2 is 1. By the statement's logic, 1 does not equal 0, and is therefore determined to be FALSE. The else statement then takes effect and prints out "5 is odd!"

Continuing down, the next instance returns a value of TRUE, since 4 MOD 2 is, in fact, 0. It then activates the code immediately after the if statement to print "4 is even!"

EXAMPLE 2: A *Really Bad Three Strikes*

The objective of the below program is to prompt a user for a number between 10 and 20. It gives the user three chances to do so. If they fail after the third chance, the program exits.

This program will be improved greatly over the course of the chapter.

```
import java.util.Scanner;  
public class Example2{  
    public static void main(String args[]){  
        //setup the Scanner  
        Scanner s = new Scanner(System.in);  
  
        System.out.println("Enter a number between 10 and 20: ");  
  
        //needed variables  
        int num = 0, strikes = 0;  
  
        //get next number from user  
        num = s.nextInt();  
  
        //first check  
        if(num < 10 || num > 20) {  
            //bad number so increase strike count  
            strikes++;  
        }else { //assumes correct number  
            System.out.println("Thank you! You entered " + num);  
            System.exit(1); //terminate  
        }  
  
        //otherwise prompt again and get number
```

```

System.out.println("Strike 1. Try again: ");
num = s.nextInt();

//second check
if(num < 10 || num > 20) {
    //bad number so increase strike count
    strikes++;
} else { //assumes correct number
    System.out.println("Thank you! You entered " + num);
    System.exit(1); //terminate
}

//otherwise prompt again and get number
System.out.println("Strike 2. Last chance: ");
num = s.nextInt();

//last check
if(num < 10 || num > 20) {
    //bad number so increase strike count
    strikes++;
} else { //assumes correct number
    System.out.println("Thank you! You entered " + num);
    System.exit(1); //terminate
}

//strike limit reached so exit
if(strikes == 3) {
    System.out.println("You can't follow directions");
    System.exit(1);
}
} //main
} //class

```

Let's see what happens here when given the below sample's output:

Enter a number between 10 and 20:

5

Strike 1. Try again:

3

Strike 2. Last chance:

10

Thank you! You entered 10

It first asks the user for the number. Here, the user entered 5, which is not what we wanted. The program checks if the entered number is out of range by the first if statement in the program. It is out of range if it is strictly less than 10 or strictly greater

than 20. We therefore increase the strike count and continue the program by asking the user again for a number.

The user then entered 3, which, once again, is not what was needed. The second if statement checking for the range of the number returns true, so, we once again increase the strike count and continue the program. This is the user's last chance.

Finally, the user gets it right by entering 10. The program prints a message accordingly. Of course, the user could have gotten it right off the bat by entering 15, in which case, the program would print the message and exit with the else clause of the first if statement.

This will be improved over the course of the chapter.

NESTED IF STATEMENTS

In Java, it is possible to *nest* an if statement inside another, and another, and another, etc. Here is a sample of that. You must be careful about which else goes with what if statement. Notice the indentation is there to help you out.

```
if(x > 0){  
    if(x != 4){  
        System.out.println("Good");  
    }else{  
        System.out.println("Bad");  
    }  
}else{  
    System.out.println("Ugly");  
}
```

Let's follow the path of these statements.

First, it checks if $x > 0$. If it returns true, it then checks if $x \neq 4$. If $x \neq 4$ returns true, "Good" is printed, otherwise "Bad" is printed. Now if $x > 0$ had returned false, "Ugly" is printed.

Or it can be something like this:

```
if(a > b) {
```

```

        if(b != c) {
            if(d >= e) {
                System.out.println("Scenario 1");
            }else{
                System.out.println("Scenario 2");
            }
        }else{
            System.out.println("Scenario 3");
        }
    }else{
        System.out.println("Scenario 4");
    }
}

```

Let's assume the following values for the variables:

$a = 3, b = 2, c = 7, d = 5, e = 6$

It first checks if a is strictly larger than b ($3 > 2$), which in this case is TRUE. Continuing down, it then checks if b is not equal to c ($2 \neq 7$), which is again TRUE. Continuing down, it then checks if d is greater than or equal to e ($5 \geq 6$), which returns FALSE, so the else statement associated with the logical check for variables d and e will activate. Therefore, the output of the above is "Scenario 2."

EXAMPLE 3: Simple Nested If

The objective of the below program is to prompt a user for a number between 10 and 20. It gives the user three chances to do so. If they fail after the third chance, the program exits.

This program will be improved greatly over the course of the chapter.

```

import java.util.Scanner;
public class Example3{
    public static void main(String args[]){
        //setup the Scanner
        Scanner s = new Scanner(System.in);

        //prompt user
        System.out.println("Enter 3 numbers, whole or decimal: ");

        //needed variables
        double a, b, c;
    }
}

```

```

//get values
a = s.nextDouble();
b = s.nextDouble();
c = s.nextDouble();

//a series of decisions
if (a > b) {
    if(a > c) {
        System.out.println("Apples");
    }else {
        System.out.println("Oranges");
    }
}else if(b > a) {
    if(b > c) {
        System.out.println("Tomato");
    }else {
        System.out.println("Potato");
    }
}

} //main
} //class

```

Let's see what happens when we have the following sample output:

```

Enter 3 numbers, whole or decimal: 5 9 3
Tomato

```

Why is that? Well, let's start at the top. We know that $a = 5$, $b = 9$ and $c = 3$. The if statement first checks if a is strictly bigger than b . Here that returns false and it then evaluates the else if statement associated with it. That statement checks if b is strictly bigger than a , which returns true.

The following statement underneath checks if b is strictly bigger than c , which is true, so Tomato prints to the screen and the program terminates.

Let's try another run through with these values:

```

Enter 3 numbers, whole or decimal: 9 4 6.3
Apples

```

Following the path of decisions, a is indeed strictly larger than b . Evaluating the next if statement, a is indeed strictly bigger than c , so Apples will print to screen and the program will terminate. But what happens with this run?

Enter 3 numbers, whole or decimal: 7.0 7.0 7.0

Yep, nothing happens. Why? Because 7.0 is not strictly larger than 7.0 and 7.0 is not strictly larger than 7.0. There is no else clause associated with the outer if statement so nothing will print to screen. The program will just terminate.

EMPTY DECISIONS

Beware of the empty decisions!!! In Java, a semicolon can be a complete statement and sometimes even the best programmers get mixed up with syntax. This happens because of the empty statement. Let's look closely at this if statement:

```
if(x == 0);  
    System.out.println((x+3));
```

Seemingly, this will print out the value of 3. However, the empty statement is seen as part of the if statement and that will be executed!!! That means, 3 will print out **no matter whether the condition is true or not**.

The corrected statement should have this single semicolon removed. Here is the corrected if:

```
if(x == 0)  
    System.out.println((x+3));
```

EXAMPLE 4: *The Empty Decision*

This program is supposed to print out the doubled version of an integer entered. However, if the number is negative, it should not be doubled. Try and determine the output before viewing it below.

```
public class Example4{
    public static void main(String args[]) {
        int x = 1;

        if(x > 0);
            System.out.println("The double of that number is: "
                + x*2);

    } //main
} //class
```

The output from running the above program is:

The double of that number is: 2

This is the case because the if statement is not executed properly. No matter whether or not the if statement returns true or false, there will always be output to the above program.

WHILE STATEMENTS

Let's say we wanted to give the user a strict rule of entering only an even number. It is simple, as it can be an if statement, but not exactly. Say that a program requires a certain input (a correct one) that even with an if statement and some code, only gives the user 1 more chance to enter that data correctly. If the user still doesn't do it right, the program will not run correctly.

This problem can be solved using what's called a **while statement**. By its definition, while a certain logical condition is **TRUE**, do something.

```
while( condition(s) ){
    //code to execute while condition(s) are true
}
```

This can also be referred to as while looping.

EXAMPLE 5: Even / Odd Detection Using While

The objective of this program is to make a user enter a **positive, even** number. It will use a while statement, as some users may try to "break the rules" of this program.

```
// checks repeatedly for an even number using while
import java.util.Scanner;
public class Example5{
    public static void main(String args[]){
        int num;
        System.out.println("Give me any positive even number: ");

        //setup the Scanner for input
        Scanner s = new Scanner(System.in);

        //get the number from the user
        num = s.nextInt();

        //as long as the number is odd, keep asking for another
        while(num % 2 == 1 || num <= 0){
            System.out.println("Not even, try again: ");
            num = s.nextInt();
        }
        //display result
        System.out.println("Right! You entered: " + num);
    } //main
} //class
```

Let's see a sample run of this program:

```
Give me any positive even number: -9
Not even or positive, try again: 3
Not even or positive, try again: 2
Right! You entered: 2
```

Let's start at the top. The first number the user entered was -9. The while condition states that the number needs to be odd or less than or equal to 0 to keep executing the while condition's code. In this case, it is true, since -9 is less than or equal to 0. It gives the user another chance to enter a correct number.

After the user enters another number, the while statement is checked again. Once again, there was an incorrect number entered of 3, so the while statement continues.

Finally, the user gets it right by entering a positive, even number of 2.

EXAMPLE 6: *Printing characters*

The objective of this program is to make a user enter any symbol and a number greater than 0. It will use a while statement to print X amount of those characters or symbols entered.

```
import java.util.Scanner;
public class Example6{
    public static void main(String args[]){
        String word = "";
        int num = 0;

        //prompt user
        System.out.println("Give me any letter or symbol: ");

        //setup the Scanner for input
        Scanner s = new Scanner(System.in);

        //get the symbol from the user
        word = s.next();

        //prompt user
        System.out.println("Enter any number > 0: ");
        num = s.nextInt();

        //while number is negative or 0, enter again
        while (num <= 0) {
            System.out.println("Try again: ");
            num = s.nextInt();
        }

        //print num number of symbols on the screen
        while(num > 0){
            System.out.print(word);
            num--; //decrease counter
        }
    } //main
} //class
```

Let's see a sample run of the above program:

Enter any letter or symbol: A

Enter any number > 0: 5

AAAAA

As you can see, 5 A symbols print on the screen, as desired. Here is another potential output:

So on and so forth. But why is that? Let's observe the while loops. In the immediate above output, the user enters 'r'. The program then asks for the user to enter any number greater than 0. Here, -9 is entered, which is incorrect. The first while statement will prompt the user to keep entering a number until the proper number is entered. Here, the user enters 48.

The next while statement will execute so long as the correctly-entered number is greater than 0. Each time through, it will print the entered 'r' symbol and decrement the num variable. It then rechecks the while condition until the num variable is 0, which will break the while statement as the condition is false ($0 > 0$). The program then terminates.

EXAMPLE 7: A Much Cleaner Three Strikes

The objective of the below program is to prompt a user for a number between 10 and 20. It gives the user three chances to do so. If they fail after the third chance, the program exits. This is a much more efficient and cleaner version of the earlier example in this chapter.

```
import java.util.Scanner;
public class Example7{
    public static void main(String args[]){
        int n = 0, strikes = 0;
        System.out.println("Please enter a positive integer: ");

        //setup the Scanner for input
        Scanner s = new Scanner(System.in);

        //get the integer from the user
        n = s.nextInt();

        //number is odd or negative
        while(n%2 == 1 || n < 0){
            //increase strike count
            strikes++;
        }
    }
}
```

```
        if(strikes == 3){  
            System.out.println("You Failed");  
            System.exit(1);  
        }  
  
        //get another value from the user  
        System.out.println("Not even or positive. Try again: ");  
        n = s.nextInt();  
    }  
    System.out.println("Thank you. You entered " + n);  
} //main  
} //class
```

A sample run of this program is:

```
Please enter a positive integer:  
-9  
Not even or positive. Try again:  
-5  
Not even or positive. Try again:  
-3  
You Failed
```

Another sample run could be:

```
Please enter a positive integer:  
-910  
Not even or positive. Try again:  
5  
Not even or positive. Try again:  
44  
Thank you! You entered 44
```

DO/WHILE STATEMENTS

A **Do/While statement** (also called a **do/while loop**) is similar to a while statement, but there is one exception: the body of the do/while statement is executed **at least once** before evaluating the while statement logic. Here is how to declare the Do/While loop:

```
do{  
    //actions here  
}while( condition(s) );
```

Notice that the while statement is at the end. This means that the test condition(s) are checked **AFTER** the code has executed **at least once**.

So take this code snippet for example. What will the output be?

```
int x = 10;  
  
do{  
    System.out.println("x= " + x);  
    x++;  
}while(x < 10);
```

The output when run would be:

x=10

Notice the value of x is printed out due to the *do* statement. The value of x then increases to 11, **and then** the while condition is checked.

EXAMPLE 8: Guess a Number

This program asks the user to pick a number between 1 and 10. If they guess right, the program terminates, otherwise they keep guessing until they get it right.

```
import java.util.Scanner;  
public class Example8{  
    public static void main(String args[]){  
        //Scanner and variables  
        Scanner s = new Scanner(System.in);  
        short entry = 0, magic_number = 7;  
  
        do {  
            System.out.println("Enter a number from 1 to 10: ");  
            entry = s.nextShort();  
        }while(entry != magic_number);
```

```
        System.out.println("You guessed it!");
    } //main
} //class
```

The program will run until the user guesses the magic number. With the use of the do statement, the user is prompted for the entry before the while condition is checked. Assuming the while statement is false, it will ask the user again for the number until they get it right.

A sample run of the above can be:

```
Enter a number from 1 to 10: 3
Enter a number from 1 to 10: 4
Enter a number from 1 to 10: 5
Enter a number from 1 to 10: 7
You guessed it!
```

BREAK & CONTINUE

With while statements (or looping in general), there are two keywords that can either stop or continue iterations. If the keyword **break** appears, the entire loop/statement is exited and the program will proceed normally. If the keyword **continue** appears, all code below that keyword will NOT be executed, and the loop will proceed with the next iteration (if there is any more left).

Here is an example with **break**:

```
public class BreakExample{
    public static void main(String args[]) {
        int i = 0;
        while(i < 15) {
            if(i == 5) break;
            System.out.println("i= " + i);
            i++;
        }
        System.out.println("Ending here...");
    } //main
} //class
```

With the above program, as soon as i is equal to 5, the loop will exit and the program will continue as normal by printing “Ending here...” No other code in the loop will be executed. The output from the above program is:

```
i = 0  
i = 1  
i = 2  
i = 3  
i = 4  
Ending here...
```

Now here is an example using **continue**:

```
public class ContinueExample{  
    public static void main(String args[]){  
        int i = 0;  
        while(i < 15){  
            if(i >= 5 && i <= 11) continue;  
            System.out.println("i= " + i);  
            i++;  
        }  
        System.out.println("Ending here...");  
    } //main  
} //class
```

With this program, when i is between 5 and 11, no code is executed below the keyword and the loop will proceed with the next iteration. The output from the above program is:

```
i = 0  
i = 1  
i = 2  
i = 3  
i = 4  
i = 12  
i = 13  
i = 14  
Ending here...
```

These keywords are used whenever they are appropriate to use in a program. Break is the most commonly used.

SWITCH/CASE STATEMENTS

There is another type of control statement in Java called a switch/case block. This is a different version of an if statement.

Here is what a general switch case block looks like:

```
switch( variable ){
    case value:
        //code for value
        break;
    default:
        //code for default
        break;
}
```

Where in the above, *variable* is a variable of an **integral type (int)**, *value* is the value of the variable at that point, and *default* is an action to take place when, and only when, all above cases are FALSE. There is no limit to how many cases you can have.

A break must occur after the actions for your case are completed, otherwise the program will not stop going to the next case(s) until it sees a break.

EXAMPLE 9: Even/Odd Detection Switch

Remembering the above program for detecting an even number with an if statement, let's change that into a switch statement:

```
import java.util.Scanner;
public class Example9{
    public static void main(String args[]){
        int num = 0;
        System.out.println("Give me any number: ");

        //setup Scanner for input
        Scanner s = new Scanner(System.in);
```

```

//get number from the user
num = s.nextInt();

//determine the value
switch(num % 2){
    case 0:
        System.out.println("It's even!");
        break;

    case 1:
        System.out.println("It's odd!");
        break;
}
} //main
} //class

```

While testing for conditions, if you want to have a certain action repeated for multiple cases, you can do the following (observe the program below).

EXAMPLE 10: *Alphabet Switch*

```

import java.util.Scanner;
public class Example10{
    public static void main(String args[]){
        String ch;
        System.out.println("Enter a letter in the alphabet: ");

        //setup the Scanner for input
        Scanner s = new Scanner(System.in);

        //get the character from the user
        ch = s.next();

        //determine its type
        switch(ch){
            case "A":
            case "a":
            case "E":
            case "e":
            case "I":
            case "i":
            case "O":
            case "o":
            case "U":
            case "u":
                System.out.println("It's a vowel");
                break;
        }
    }
}

```

```

        default:
            System.out.println("It's a consonant.");
            break;
    }
} //main
} //class

```

Note the beginning of the switch block with the cases. All of those cases have the same action, so it is perfectly legal to write them down on the line as above.

Notice also that it is assumed here that the default case is used for a consonant when in reality, it could be anything such as a number or other character. For now, it is a safe assumption that the user follows directions.

EXAMPLE 11: *Zodiac Signs*

This program will prompt the user to enter their birthday in a particular format of month (1 to 12) and day they were born (1 to 31). The program will do some error checking to be sure the entries are correct. If not, they prompt the user for a corrected entry until they get it right. Afterwards, it will determine what zodiac sign you are based on the date you entered. The signs of the zodiac are:

SIGN	STARTS	ENDS
Aquarius	January 20	February 18
Pisces	February 19	March 20
Aries	March 21	April 19
Taurus	April 20	May 20
Gemini	May 21	June 20
Cancer	June 21	July 22
Leo	July 23	August 22
Virgo	August 23	September 22
Libra	September 23	October 22
Scorpio	October 23	November 21
Sagittarius	November 22	December 21
Capricorn	December 22	January 19

```

import java.util.Scanner;
public class Example11{
    public static void main(String args[]) {

```

```

//Scanner for input
Scanner s = new Scanner(System.in);

//variables for month and day
int m = 0, d = 0;

//get month from user
System.out.println("Enter month (1 to 12): ");
m = s.nextInt();

//while month is out of range, prompt again
while (m < 1 || m > 12) {
    System.out.println("Enter month again: ");
    m = s.nextInt();
}

//use a switch on the month...
//30 days as September, April, June and November
//all the rest have 31
//except February which has 28 and 29 on a leap year
switch(m) {
    case 9:
    case 4:
    case 6:
    case 11:
        //month that has 30 days
        System.out.println("Enter day (1 to 30): ");
        d = s.nextInt();

        //while day out of range
        while (d < 1 || d > 30) {
            System.out.println("Enter day again: ");
            d = s.nextInt();
        }
        break;

    case 2:
        //month has up to 29 days
        System.out.println("Enter day (1 to 29): ");
        d = s.nextInt();

        //while day out of range
        while (d < 1 || d > 29) {
            System.out.println("Enter day again: ");
            d = s.nextInt();
        }
        break;

    default:
        //month has 31 days
        System.out.println("Enter day (1 to 31): ");
}

```

```

d = s.nextInt();

        //while day out of range
        while (d < 1 || d > 31) {
            System.out.println("Enter day again: ");
            d = s.nextInt();
        }
    }

    //have month and day so figure out sign.
    if((m==1 && d>=20) || (m==2 && (d>=1 && d<=18))) {
        System.out.println("You are an Aquarius!");

    }else if((m==2 && d>=19 ) || (m==3 && (d>=1 && d<=20))) {
        System.out.println("You are a Pisces!");

    }else if((m==3 && d>=21) || (m==4 && (d>=1 && d<= 19))) {
        System.out.println("You are a Aries!");

    }else if((m==4 && d>= 20) || (m==5 && (d>=1 && d<=20))) {
        System.out.println("You are a Taurus!");

    }else if((m==5 && d>=21) || (m==6 && (d>=1 && d<=20))) {
        System.out.println("You are a Gemini!");

    }else if((m==6 && d>=21) || (m==7 && (d>=1 && d<=22))) {
        System.out.println("You are a Cancer!");

    }else if((m==7 && d>=23) || (m==8 && (d>=1 && d<=22))) {
        System.out.println("You are a Leo!");

    }else if((m==8 && d>=23) || (m==9 && (d>=1 && d<=22))) {
        System.out.println("You are a Virgo!");

    }else if((m==9 && d>=23) || (m==10 && (d>=1 && d<=22))) {
        System.out.println("You are a Libra!");

    }else if((m==10 && d>=23) || (m==11 && (d>=1 && d<=21))) {
        System.out.println("You are a Scorpio!");

    }else if((m==11 && d>=22) || (m==12 && (d>=1 && d<=21))) {
        System.out.println("You are a Sagittarius!");

    }else if((m==12 && d>=22) || (m==1 && (d>=1 && d<=19))) {
        System.out.println("You are a Capricorn!");
    }

} //main
} //class

```

This program shows numerous ways to make a decision. It uses while statements to keep getting data from the user, should it be wrong; a switch statement to figure out which month has what amount of days; and a series of if statements to determine what the zodiac sign is.

A sample run of the above program can be:

```
Enter a month (1 to 12): 9
```

```
Enter a day (1 to 30): 10
```

```
You are a Virgo!
```

EXERCISES

DIRECTIONS: Answer each question below to the best of your ability. Solutions on page 590.

Problem 1:

What is the output of each of the following programs?

a)

```
public class A{
    public static void main(String args[]) {
        int q = 4;
        if(q <= 5) System.out.println("Yes!");
        else System.out.println("Nope.");
    } //main
} //class
```

b)

```
public class B{
    public static void main(String args[]) {
        int q = 43;
        if(q % 5 == 0) System.out.println("Ends in 5");
        else System.out.println("Doesn't end in 5");
    } //main
} //class
```

c)

```
public class C{
    public static void main(String args[]) {
```

```

boolean q = true;

if(!q) System.out.print("Yes!");
else System.out.print("Nope.");

} //main
} //class

d)
public class D{
    public static void main(String args[]){

        int q = -4;

        if(q != 0 && (q < -3) ) System.out.print("Yes!");
        else System.out.print("Nope.");

    } //main
} //class

e)
public class E{
    public static void main(String args[]){

        int q = 423;

        if(q/10 > 5) System.out.print("Bigger!");
        else System.out.print("Not bigger.");

    } //main
} //class

f)
public class F{
    public static void main(String args[]){

        int q = 4, j = q % 3;

        if(j == 1 || q < 5 && j+q >= 5)
            System.out.print("Yes!");
        else
            System.out.print("Nope.");

    } //main
} //class

g)
import java.util.Scanner;
public class G{
    public static void main(String args[]){
        int x = 0, y = 0;

```

```

        System.out.println("Please enter x and y: ");
        Scanner s = new Scanner(System.in);

        x = s.nextInt();
        y = s.nextInt();

        if(x+y == 10 || x%y == 1) System.out.println(x);
        else System.out.println(y);

    } //main
} //class

```

1. What is the output when the user enters 5 for x and 8 for y?

2. What is the output when the user enters 10 for x and 6 for y?

```

h)
import java.util.Scanner;
public class H{
    public static void main(String args[]){
        int x = 0, y = 0;

        System.out.println("Please enter x and y: ");
        Scanner s = new Scanner(System.in);

        x = s.nextInt();
        y = s.nextInt();

        if((x*y) % 7 == 0) System.out.println((x*y));
        else System.out.println((y - x + 1));

    } //main
} //class

```

1. What is the output when the user enters 15 for x and 12 for y?

2. What is the output when the user enters 12 for x and 16 for y?

```

i)
import java.util.Scanner;
public class I{
    public static void main(String args[]){
        int x = 0, y = 0;

        System.out.println("Please enter x and y: ");
        Scanner s = new Scanner(System.in);

        x = s.nextInt();
        y = s.nextInt();

```

```

        if((x+y-1) % (y-1) == 10) System.out.println((y-x+1));
        else System.out.println("2");

    return 0;
}

```

1. What is the output when the user enters 11 for x and 11 for y?

2. What is the output when the user enters 20 for x and 26 for y?

```

j)
import java.util.Scanner;
public class J{
    public static void main(String args[]){
        int x = 0, y = 0;

        System.out.println("Please enter x and y: ");
        Scanner s = new Scanner(System.in);

        x = s.nextInt();
        y = s.nextInt();

        if(x%2 == 0)
            if(y % 2 == 1) System.out.println("Even Odd");
            else System.out.println("Even Even");
        else
            if(y % 2 == 1) System.out.println("Odd Odd");
            else System.out.println("Odd Even");

    } //main
} //class

```

1. What is the output when the user enters 5 for x and 5 for y?

2. What is the output when the user enters 3 for x and 4 for y?

Problem 2:

Given the following program:

```

public class Problem2{
    public static void main(String args){

        long x = 0;
        int s = 104, times = 1;

        Scanner sc;
        System.out.println("Enter a number: ");

```

```

sc = new Scanner(System.in);

x = sc.nextInt();

while(x != s){
    ++times;

    if(x < s){
        System.out.println("Enter a number HIGHER:");
    }else if(x > s){
        System.out.println("Enter a number LOWER:");
    }
}

sc = new Scanner(System.in);
x = sc.nextInt();

}

System.out.println("You took " + times +
    " tries to get it right!");

System.out.println("The secret number is: " + s);
} //main
} //class

```

What is the output when the number entered first is 104?

Problem 3:

Write a while loop to perform each of the following tasks:

- Print out the first 25 odd numbers.
- Print out the '^' character 10 times.
- Find the sum of the first 100 even numbers.
- Print out the first 20 numbers that end with a 5.
- Print out 5 rows of the '#' character. Each row contains 10 characters.
- Print out a table of 3 rows and 2 columns, where the numbers printed are the product of the row and column counters (begin at the value of 1 for each counter).

Problem 4:

Given each while loop below, what is the output.

a)

```

int i = 0;
while(i < 10){
    System.out.print(i++);
}

```

```
b)
int x = 6;
while(x % 2 == 0){
    System.out.print(x++);
}

c)
int q = 0, p = 1;
while(p == 1){
    System.out.print((q + 2));
    q += 2;
    if(q == 14) p = 0;
}

d)
int i = 9, j = 3;
while(j != 0){
    System.out.print(--i);
    j = i;
}

e)
int x = 20;
while(x > 9){
    System.out.print(x*2 + "*");
    x -= 2;
}

f)
int z = 0, zz = 9, zzz = 18;
while(zz < zzz){
    System.out.print(z++);
    zz++;
}

g)
double x = 0.0;
while(x != 2.0){
    x = x + 0.5;
    System.out.print((x+x));
}

j)
long x = 9000, i = 10;
while(x/100 != 0){
    System.out.print(x % i + "-");
    x = x / 100;
}
```

```
k)
int x = 8;
while(x > 0) {
    System.out.print(--x*10);
    x = x-1;
}

l)
double g = 9.9;
while(g > 8.8) {
    System.out.println("G: " + g-0.1);
    g = g - 0.1;
}

m)
int s = 11;
int p = s;
while(p % 3 != 1) {
    System.out.print((p+2));
    p = p+1;
}
```

CHAPTER 6

Strings in Java

This chapter covers one of the most used classes in Java called String. It shows how to use and manipulate String objects with the use of many member methods (substring, indexOf, toUpperCase, toLowerCase), etc...

TOPICS

1. <i>Introduction</i>	89
2. <i>String Equalities</i>	93
3. <i>Comparing Strings</i>	95
4. <i>Concatenating Strings</i>	99
5. <i>More Examples</i>	100
6. <i> StringTokenizer Class</i>	106
7. <i>Exercises</i>	110

In Java, but unlike C++, the string data type is now an object type. Notice that the class name starts with an **uppercase** (String). If you do not capitalize this, Java will not know what you are referring to.

By its conception, a **String** is anything and everything grouped together between two double quotes. For example, "Hello world" is one string while "sdad anesd ccn" is another or "The car broke down on the way to the wedding!" or even "123456 78910."

For the purpose of some examples below, characters in a String have what's called an **index**, or position, in the String as a whole. For instance, take the String "Hello." The numbers above the grid represent each character's index in the String.

0	1	2	3	4
H	e	l	l	o

Another important thing to note is that white space counts in a String! Take for instance the String "Hello there":

0	1	2	3	4	5	6	7	8	9	10
H	e	l	L	o		t	h	e	r	e

Note that the index of the white space character is 5.

Strings can also be manipulated in numerous ways. The first is with some **member methods** of the String class. Let's see an example first, before we continue (methods will be discussed in much more detail in Chapter 8).

EXAMPLE 1: *String Methods Example*

```
public class Example1{
    public static void main(String args[]) {
        String word;

        //assign the string to the variable:
        word = "Alexander";

        //1. retrieve the length by calling the
        //length method:
        int length = word.length();
        System.out.println("Length: " + length);
```

```

//2. use the case methods:
System.out.println("toUpperCase: " + word.toUpperCase());
System.out.println("toLowerCase: " + word.toLowerCase());

//3. use the trim method to eliminate leading
//or trailing white spaces:
word = word.trim();
System.out.println("trim: " + word);

//4. check for a certain character using indexOf()
System.out.println("indexOf('s'): " + word.indexOf('s'));

//5. print out the beginning character using charAt()
System.out.println("first character: " + word.charAt(0));

//6. make the string shorter
word = word.substring(0, 4);
System.out.println("shorter string: " + word);

} //main
} //class

```

A lot has happened in just a short program. Let's see the output when the program is run:

```

Length: 9
toUpperCase: ALEXANDER
toLowerCase: alexander
trim: Alexander
indexOf('s'): -1
first character: A
shorter string: Alex

```

Now let's observe some of the methods that were used above:

int length()

The length() method will simply return the length of a string as an integer. Say the string was "Hello World." This method would return a value of 11.

String toUpperCase()

The toUpperCase() method will return the uppercase version of a string. Say you have a string "Welcome." This method will return "WELCOME." Only the letters a-z and A-Z are affected; not numbers or symbols.

String toLowerCase()

The `toLowerCase()` method will return the lowercase version of a string. Say you have a string "Welcome TO Earth." This method will return "welcome to earth." Only the letters a-z and A-Z are affected; not numbers or symbols.

String trim()

The `trim` method will return the string without leading or trailing white space characters. Say that a string was " hello ". There are 4 spaces in the front and 3 spaces at the end. The `trim` method would make this "hello."

char charAt(int index)

This method will look for a character at a specific index. It will return that character if it is found. Say we have the string "Hello" and we say `charAt(4)`. It will return 'o' as that character is at index 4.

```
int indexOf(int ch)
int indexOf(int ch, int begin)
int indexOf(String ch)
int indexOf(String ch, int begin)
```

Notice that there are 4 different methods listed here. All of them perform the same overall action of returning an integer, which represents the **FIRST OCCURRANCE** of a character or String contained in that string. So say that we have the string "Hello" and we say `indexOf('l')`. This method will use the first method above and return 2. Notice it doesn't return 3 as the **first occurrence** of the character l is at index 2. We can also say, using the same string "Hello", `indexOf("He")`. It will return 0 as the string that you are searching for starts at index 0.

By default, if a string or character is not found in the string, any of the methods will return -1.

String substring(int begin)**String substring(int begin, int end)**

Either of these methods will shorten a string when called. If no ending index is specified, it will return the rest of that string. Say we have the string "Hello there" and we say `substring(4)`, the method will return "o there." Let's use `substring(2,5)`, the method will return "llo."

Let's take some Strings and expand on the above methods. What will be the output of the below using the two Strings below?

```
String word1 = "ABcd eFG";
String word2 = "1234567";
```

- a) word1.length();
- b) word2.length();
- c) word1.toUpperCase();
- d) word2.toLowerCase();
- e) word2.charAt(5);
- f) word1.charAt(5);
- g) word1.indexOf('f');
- h) word1.indexOf('B');
- i) word2.indexOf('9');
- j) word2.substring(3);
- k) word2.substring(word1.length()-4);
- l) word1.substring(0, 3);

So, let's see each of the above and explain.

a) word1.length();

This would return a value of **8**. The length() method simply accounts for the actual length of the String.

b) word2.length();

This would return a value of **7**. Once again, the length() method simply accounts for the actual length of the String.

c) word1.toUpperCase();

This would return a value of "ABCD EFG." The toUpperCase() method simply returns the upper case version of the String.

d) word2.toLowerCase();

This would return a value of "1234567." The toLowerCase() method simply returns the lower case version of the String. As noted earlier, numbers and symbols are not affected by either the toUpperCase() or toLowerCase() method.

e) word2.charAt(5);

This would return **5**. The charAt() method simply returns a character at a particular index of the String.

f) word1.charAt(5);

This would return [white space] as a blank character appears at index 5 of the String.

g) word1.indexOf('f');

This would return -1 as there is no 'f' in the String. The indexOf() method returns the first occurrence of the searched for character in the String.

h) word1.indexOf('B');

This would return 1 as this is the first occurrence of 'B' in the String. The `indexOf()` method returns the first occurrence of the searched for character in the String.

i) `word1.indexOf('9');`

This would return -1 as there is no '9' in the String.

j) `word2.substring(3);`

This would return "4567" as the `substring()` method returns a portion of the String beginning a specified index. If no ending index is given (as is the case here), it continues until the end.

k) `word2.substring(word1.length()-4);`

First, we need to determine what the length of `word1` is. As noted above, it is 8. Therefore, the starting index is 4 as $8-4 = 4$. This would return "567."

l) `word1.substring(0, 3);`

This would return "ABC." As noted earlier, when an ending index is specified, it returns a String starting at the given index and ending at the given index.

STRING EQUALITIES

Java provides some methods that will check to see if two Strings are equal. To show this, here is a small program that will check a certain String entered by the user against a keyword. It gives the user 3 attempts to get the word correct.

EXAMPLE 2: *String Equalities*

```
import java.util.Scanner;
public class Example2{
    public static void main(String args[]){
        //needed variables
        Scanner s = new Scanner(System.in);
        String keyword = "Balloon", entry = "";

        //prompt user
        System.out.println("Enter a word: ");

        //get the character from the user
        entry = s.nextLine();

        //give user 3 chances to enter the keyword
        int chances = 1;
```

```

        while (chances < 3) {
            if(entry.equals(keyword)) {

                //found the word exactly as typed
                //print message and exit program
                System.out.println("You found it exactly!");
                System.exit(1);

            }else if (entry.equalsIgnoreCase(keyword)) {

                //found it but not exactly as case is ignored
                //print message and exit
                System.out.println("Found it! Case was ignored!");
                System.exit(1);

            }else {
                //max attempts reached
                if(chances == 3)
                    break;

                //no match so try again
                System.out.println("Try again. " + (3-chances)
                    + " chances left: ");

                //get String from user
                entry = s.nextLine();

                //no match. Try again
                chances++;
            }
        }

        //no successful entries if you arrived here
        System.out.println("You have failed after 3 attempts");
    } //main
} //class

```

Let's see a sample output of the above program:

```

Enter a word: Red
Try again. 2 chances left: Yellow
Try again. 1 chances left: Balloon
You found it exactly!

```

Here's another sample run:

```
Enter a word: Tired  
Try again. 2 chances left: BALLoon  
Found it! Case was ignored!
```

And one more:

```
Enter a word: One  
Try again. 2 chances left: Two  
Try again. 1 chances left: Three  
You have failed after 3 attempts
```

Let's discover why. Here are the descriptions of the methods used:

boolean equals(String another)

The `equals()` method will take another `String` as its parameter and check if it is **EXACTLY** like the initial `String`, where initial `String` is what you called the method from.

This method returns a boolean value (either true or false). If true, the `String` parameter is an **exact copy**, character for character, of the initial `String`. If false, the argument is not a copy as something differs in the `String`.

boolean equalsIgnoreCase(String another)

This method is similar to the regular `equals()` method with the exception of this method not taking into account the case (either upper or lower) of BOTH `Strings`.

Strings will be used intermittently throughout the book in some example programs and chapter exercises.

COMPARING STRINGS

There are times when we need to compare two `String` objects to determine which one comes before or after the other when sorting in alphabetical order (or **lexicographically**). This can be accomplished with the `compareTo()` and `compareToIgnoreCase()` methods:

int compareTo(String another)

The `compareTo()` method compares two `Strings` lexicographically, via its Unicode values (or ASCII values (see Appendix A)). The method returns an integer value that is negative if the primary `String` is lexicographically before the argument `String`; 0 if they are exactly equal; and a positive value if the primary `String` is lexicographically after the argument `String`.

int compareToIgnoreCase(String another)

This method is similar to the above `compareTo()` method with the exception of this method not taking into account the case (either upper or lower) of BOTH `Strings`.

Let's see a short example of the above. Say that we have the following code snippet:

```
String word1 = "Orange", word2 = "Alexander", word3 = "Alex";
System.out.println(word1.compareTo(word2));
System.out.println(word2.compareTo(word3));
System.out.println(word3.compareTo(word2));
```

The output would be:

```
14
5
-5
```

For the first line of output, lexicographically, the characters differ in the first position. It compares O and A, which have a difference of 14. That means that Orange comes after Alexander, as the value is positive.

For the second line of output, the characters differ in the fifth position. In the case when one `String` is larger than the other, the shorter `String`, lexicographically, comes before the longer `String`.

For the final line of output, it is the reverse of the above, since the shorter `String` is the primary `String` being compared to the longer one.

EXAMPLE 3: Comparing Strings

This example will ask the user to enter 4 `String` values and perform some comparisons on them.

```

import java.util.Scanner;
public class Example3{
    public static void main (String args[]) {
        Scanner s = new Scanner(System.in);
        System.out.println("Enter 4 words separated by a space: ");

        //get the 4 words from the console
        String w1,w2,w3,w4;
        w1 = s.next();
        w2 = s.next();
        w3 = s.next();
        w4 = s.next();

        //declare and obtain different comparison values
        int v1, v2, v3, v4;
        v1 = w1.compareTo(w2);
        v2 = w2.compareTo(w3);
        v3 = w3.compareTo(w4);
        v4 = w4.compareToIgnoreCase(w1);

        //print the 4 values
        System.out.println("V1: " + v1 + " V2: "
                           + v2 + " V3: " + v3 + " V4: "
                           + v4);

        if(v1 > v2) w1 = w3;
        if(v3 < v4) w2 = w4;
        if(v2 < v4) w3 = w4;
        if(v4 > v1) w4 = w2;

        //print the 4 words
        System.out.println("W1: " + w1 + " W2: "
                           + w2 + " W3: " + w3 + " W4: "
                           + w4);
    } //main
} //class

```

The output from running the above program is:

Enter 4 words, each separated by a space:

Apple ORANGE Fruit snake

V1: -14 V2: 9 V3: -45 V4: 18

W1: Apple W2: snake W3: snake W4: snake

Let's see why. To begin, we get the 4 words from the user. In this case, before we begin comparing them, they are:

w1 = Apple; w2 = ORANGE; w3 = Fruit; w4 = snake;

The first comparison we make is between w_1 and w_2 . This produces a value of -14 for the v_1 variable, as Apple and ORANGE differ in the first character. Apple, being the primary String, comes before ORANGE, hence why the value is negative.

The next comparison we make is between w_2 and w_3 . This produces a value of 9 for the v_2 variable, as ORANGE and Fruit differ in the first character by 9. ORANGE, being the primary String, comes after Fruit, hence why the value is positive.

The next comparison we make is between w_3 and w_4 . This produces a value of -45 for the v_3 variable, as Fruit and snake differ in the first character. Fruit, being the primary String, comes before snake (as capital letters always precede lowercase letters), hence why the value is negative.

The final comparison we make is between w_4 and w_1 , this time ignoring its case. This produces a value of 18 for the v_4 variable. It compares snake and apple (notice lowercase A). This means snake comes after apple, hence why the value is positive.

After we obtain the values for each of the int variables, we do some comparisons between them. The current values for the variables are:

$v_1 = -14$; $v_2 = 9$; $v_3 = -45$; $v_4 = 18$

The if statements can be observed as follows:

```
if(-14 > 9) w1 = w3;  
if(-45 < 18) w2 = w4;  
if(9 < 18) w3 = w4;  
if(18 > -14) w4 = w2;
```

In order, the first if statement is false, so no actions are taken. The second if statement is true, so the new value of w_2 is the current w_4 (snake). The third if statement is true, so the new value of w_3 is the current w_4 (snake). The final if statement is also true, so the new value of w_4 is the current value of w_2 (snake).

The last System.out statement prints the values of the String variables.

Another sample run of the above is:

Enter 4 words, each separated by a space:

BUS bus TRAIN train

V1: -32 V2: 14 V3: -32 V4: 18

W1: BUS W2: train W3: train W4: train

CONCATENATING STRINGS

Java allows you to **concatenate** two or more Strings, or to put it another way, join two or more Strings together. This is done with the use of the + or += operators. Take for example this code snippet:

```
String first = "Alex", last = "Maureau", name = "";
name = first + " " + last;
System.out.println(name);
```

To put it simply, the above will concatenate the String variables first, and then output the full name. Now what if we did this?

```
String first = "Alex", last = "Maureau", name = "";
name += first + last;
System.out.println(name);
```

The output would be “AlexMaureau” without the space in between. Notice that we used the += operator for shorthand, but it could have been written like this, as well:

```
String first = "Alex", last = "Maureau", name = "";
name = name + first + last;
System.out.println(name);
```

EXAMPLE 4: Reversing a String

This example will ask the user to input a String and print the reverse of it to the screen.

```
import java.util.Scanner;
public class Example4{
    public static void main(String args[]){
        Scanner s = new Scanner(System.in);
```

```

String entry = "", reverse = "";

//get the entry from the user
System.out.println("Enter a word or phrase: ");
entry = s.nextLine();

//start the counter with the length of the entry
int counter = entry.length();

//build the reverse of the String
while(counter > 0) {
    reverse += entry.charAt(counter-1);
    counter--;
}

//print the reverse
System.out.println("The reverse of the entry is: "
    + reverse);

} //main
} //class

```

Let's see a sample output of the above program:

```

Enter a word or phrase: Hello world
The reverse of the entry is: dlrow olleH

```

And another:

```

Enter a word or phrase: 1 2 3 4 5 5 4 3 2 1
The reverse of the entry is: 1 2 3 4 5 5 4 3 2 1

```

The program asks for a String from the user. It then will reverse the String entered by concatenating the *reverse* String variable with the character at the *counter* index. The *counter* variable is initialized to the length of the String and counts down each time.

MORE EXAMPLES

In the second sample output above, this is what is called a **palindrome**. By definition, a palindrome is a word or phrase that reads the same forward and backwards. Some common palindromes in the English language are: kayak, civic, reviver, rotor, “rise to vote sir” and “never odd or even.”

EXAMPLE 5: *Check for a Palindrome*

This program will check to see if the user has entered a palindrome.

```
import java.util.Scanner;
public class Example5{
    public static void main(String args[]){
        //String variable
        Scanner s = new Scanner(System.in);
        String entry = "";

        //get the entry from the user
        System.out.println("Enter a word or phrase: ");
        entry = s.nextLine();

        //start the counter with the length of the entry
        int counter = entry.length();
        int left = 0, right = counter-1;

        //find the midpoint where the movement should stop
        int midpoint = counter/2;
        boolean isPal = true;

        //so long as we are less than the midpoint
        while(left < midpoint) {
            //two characters match. One at the left
            //and one at the right so continue
            if(entry.charAt(left) == entry.charAt(right)) {
                //move left and right indices
                left++;
                right--;
            } else {
                //mismatched characters so palindrome is false
                isPal = false;
                break; //no need to continue
            }
        }//while
        //check for the palindrome
        if(isPal) {
            System.out.println("You entered a palindrome!");
        } else {
            System.out.println("You didn't enter a palindrome!");
        }
    } //main
} //class
```

Let's see a sample run of the above program:

```
Enter a word or phrase: kayak
You entered a palindrome!
```

Or this:

```
Enter a word or phrase: street
You didn't enter a palindrome!
```

The program first sets the *counter* to the length of the entered String. It then initializes two variables, named *left* and *right*, to 0 and the (*counter*-1), respectively. This is the index of the first and last character in the String initially.

Once the program continues, it determines the midpoint of the String entered by taking the *counter* and dividing it by 2. This will always produce a whole number. So say the String entered was "Hello." The value of the *counter* variable would be 2: the length of the String is 5 and half of 5 is 2.5, but due to the variable type of integer, it will store the value 2. If the String were "Orange," the length would be 6, so the midpoint is 3.

Next, we have a boolean variable named *isPal*, which we initialize to true. At this point, it can be assumed the user has entered a palindrome, and we must prove they did not.

For the while statement, so long as the *left* variable is less than the *midpoint* variable, keep executing the code. We then check if the two characters in the entered String are equal by getting the characters at the *left* and *right* values. Assuming they are equal, we want to keep moving left and right across the String, so increment and decrement, respectively. If at any point the two characters do not match, then the user did not enter a palindrome. We can set the *isPal* variable to false, and we can then break out of the while statement.

However, there are two flaws with this program! Let's see two sample outputs and determine how we can improve it.

```
Enter a word or phrase: kayaK
You didn't enter a palindrome!
```

And this one:

```
Enter a word or phrase: rise to vote sir
```

You didn't enter a palindrome!

Both of these are, in fact, palindromes. The program can be improved in two ways:

1. Convert the entered String into either all lower case or upper case.
2. Eliminate all white spaces within the String.

Item 1 above goes with the first sample output while item 2 goes with the second. So, let's see an improved program to handle this:

EXAMPLE 6: Improved Palindrome Check

```
import java.util.Scanner;
public class Example6{
    public static void main(String args[]){
        Scanner s = new Scanner(System.in);
        String entry = "", clean = "";

        //get the entry from the user
        System.out.println("Enter a word or phrase: ");
        entry = s.nextLine();

        entry = entry.toUpperCase(); //convert to all the same case

        //get the String, free of any white spaces
        int counter = 0;
        while (counter < entry.length()) {
            //if there is not a white space add to clean String
            if(entry.charAt(counter) != ' ') {
                clean += entry.charAt(counter);
            }
            counter++;
        }

        //start counter with the length of the cleaned String
        counter = clean.length();
        int left = 0, right = counter-1;

        //find the midpoint where the movement should stop
        int midpoint = counter/2;
        boolean isPal = true;

        //so long as we are less than the midpoint
        while(left < midpoint) {
            //two characters match. One at the left
            //and one at the right so continue

```

```

        if(clean.charAt(left) == clean.charAt(right)) {
            //move left and right indices
            left++;
            right--;
        } else {
            //mismatched characters so palindrome is false
            isPal = false;
            break; //no need to continue
        }
    } //while

    //check for the palindrome
    if(isPal) {
        System.out.println("You entered a palindrome!")
    } else {
        System.out.println("You didn't enter a palindrome!");
    }
} //main
} //class

```

Seeing the two sample runs before, let's try to run the program again with those words entered as such:

```

Enter a word or phrase: kayaK
You entered a palindrome!

```

And this one:

```

Enter a word or phrase: rise to VOTE sir
You entered a palindrome!

```

Success! The revised program will convert the entered String to uppercase, in addition to removing any white spaces to allow for a foolproof check of a palindrome.

EXAMPLE 7: Pattern Printing

The below program will print a pattern to the screen, based on the entry from the user.

```

import java.util.Scanner;
public class Example7{
    public static void main(String args[]){
        Scanner s = new Scanner(System.in);
        String entry = "";

```

```

//get the entry from the user
System.out.println("Enter a word or phrase: ");
entry = s.nextLine();

//start the counter with the length of the entry
int counter = entry.length();
int printer = 0;
char temp;

while(counter > 0) {
    //if counter is even, print upper case
    //otherwise print lower case
    if(counter % 2 == 0) {
        temp = Character.toUpperCase(
            entry.charAt(counter-1));
    } else {
        temp = Character.toLowerCase(
            entry.charAt(counter-1));
    }
    //reset the printer variable
    printer=0;

    //now print the character a certain number of times
    while(printer < counter) {
        System.out.print(temp);
        printer++;
    }
    //decrease the counter variable
    counter--;
    System.out.println();
}
} //main
} //class

```

Let's see a sample run of the above program:

```

Enter a word or phrase: FIVE
EEEE
VVV
II
f

```

And another:

```

Enter a word or phrase: Apples Oranges

```

```
SSSSSSSSSSSSSS  
eeeeeeeeeeee  
GGGGGGGGGGGGG  
nnnnnnnnnnnn  
AAAAAAAAAAA  
rrrrrrrrr  
OOOOOOOO
```

```
SSSSS  
eeee  
LLLL  
ppp  
PP  
a
```

Let's go through the above program step by step. After the program prompts the user for an entry, it then initializes the *counter* variable to the length of the String. The outer while statement will return true as long as the counter variable is strictly greater than 0.

Inside the outer while statement, we first check if the *counter* variable is odd or even. If it's even, we want to assign, to the variable *temp*, an uppercase character. We make use of the Character wrapper class, as mentioned in Chapter 3. Conversely, if the counter variable is odd, we will assign a lowercase character to the variable *temp*.

Once we have that complete, the second while statement begins. Here, this will print X number of characters to the screen. Each time we print, we increment the *printer* variable by 1. This while statement will return true so long as the variable *printer* is strictly less than variable *counter*. If it is, print another character and evaluate again.

Once that while statement is finished, we decrement the *counter* variable and go back to the outer while statement to evaluate again. And so on, and so forth.

STRINGTOKENIZER CLASS

In Java, there is a way to break up a line of text into what are called tokens. A **token** is a smaller piece of a string. This method of breaking up tokens are come from the **StringTokenizer** class.

To use the StringTokenizer class, you must import it from the java.util library:

```
import java.util.StringTokenizer;
```

Most generally, here is how to declare a StringTokenizer object:

```
StringTokenizer name = new StringTokenizer(variable, token(s));
```

where in the above, *name* is an appropriate name for the tokenizer object; *variable* is the name of a String variable you are using, or even just a simple string; and *token(s)* is the token(s) you are specifying.

In the above definition, if the second argument of the constructor is not specified, the token, **by default, is a white space**. Also, if the token that is declared does not exist in the String you are searching through, the entire string is returned to you.

Let's say that we wanted to break up the following line of text into smaller pieces:

```
"Hello and welcome to programming"
```

The StringTokenizer class has some useful methods when tokenizing Strings. Here is some information below.

boolean hasMoreTokens()

This method will return a boolean value that will represent if there are any more tokens left in the line you are trying to tokenize. The true return will show that there is at least 1 more token in the line.

int countTokens()

This method will simply return an integer representing the number of tokens in the String. This method does not advance the position in the String.

String nextToken()

This method is the primary method of the class as it will get you the next piece of data in the String. The position in the string is changed after this method is called.

EXAMPLE 8: StringTokenizer Basics

This program will simply take a String and break it up into smaller pieces using a simple StringTokenizer.

```
import java.util.StringTokenizer;
public class Example8{
    public static void main(String args[]){
        String s = "What on earth is going on here?";

        //by default, a white space:
        StringTokenizer st = new StringTokenizer(s);

        //when there are still more tokens, print out the next one:
        while(st.hasMoreTokens())
            System.out.println(st.nextToken());
    } //main
} //class
```

The output from the above program is simply:

```
What
on
earth
is
going
on
here?
```

EXAMPLE 9: StringTokenizer Basics II

This program will ask the user to enter 4 numbers, that are separated by a space or a comma. The program checks if there are less than 4 numbers entered by counting the number of tokens. If it is less than 4, the program exits. Otherwise, it tokenizes the entry and calculates the sum of the numbers.

```
import java.util.StringTokenizer;
import java.util.Scanner;
public class Example9{
```

```

public static void main(String args[]){
    Scanner s = new Scanner(System.in);

    String entry;
    short n1, n2, n3, n4;

    //get input from user
    System.out.println("Enter 4 numbers separated by " +
        "a comma or space: ");
    entry = s.nextLine();

    //set the tokenizer for either a space or a comma
    StringTokenizer st = new StringTokenizer(entry, ", ");

    //not enough numbers entered so exit
    if(st.countTokens() < 4)
        System.exit(1);

    //get the entered numbers and parse them
    n1 = Short.parseShort(st.nextToken());
    n2 = Short.parseShort(st.nextToken());
    n3 = Short.parseShort(st.nextToken());
    n4 = Short.parseShort(st.nextToken());

    System.out.println("The sum is: " + (n1+n2+n3+n4));
} //main
} //class

```

The output from the above program can be:

Enter 4 numbers separated by a comma or space:

5, 5, 6, 4

The sum is: 20

Or, it can be this (notice the white space in the entry line):

Enter 4 numbers separated by a comma or space:

5, 5, 6, 4

The sum is: 20

EXERCISES

DIRECTIONS: Answer each question below to the best of your ability. Solutions on page 592.

Problem 1:

Briefly describe the action for each of the following member methods of the String class:

- a) toUpperCase()
- b) toLowerCase()
- c) trim()
- d) charAt()
- e) indexOf()
- f) equals()
- g) equalsIgnoreCase()
- h) substring()
- i) compareTo()
- j) compareToIgnoreCase()

Problem 2:

Briefly describe the action for each of the following member methods of the StringTokenizer class:

- a) hasMoreTokens()
- b) nextToken()
- c) countTokens()

Problem 3:

Using the following String below:

```
String s = "Professor";
```

What is the result of the following lines of code?

- a) s.toUpperCase()
- b) s.toLowerCase()
- c) s.charAt(4)
- d) s.charAt(8)
- e) s.indexOf('s')
- f) s.indexOf('r')
- g) s.substring(3)
- h) s.substring(3, 3)

Problem 4:

Using the following String below:

```
String s = "RevieWING Java";
```

What is the result of the following lines of code?

- a) s.toUpperCase()
- b) s.toLowerCase()
- c) s.charAt(7)
- d) s.charAt(9)
- e) s.indexOf("w")

- f) s.indexOf("Java")
- g) s.substring(5, 4)
- h) s.substring(10)

Problem 5:

What is the output of the below program?

```
public class Problem5 {  
    public static void main(String args[]) {  
        String a = "Apples", b = "Oranges";  
        System.out.println(a);  
        System.out.println(b.toUpperCase());  
        System.out.println(a.indexOf("le"));  
        System.out.println(b.substring(1, 5));  
        System.out.println(a == b);  
    }  
} //class
```

Problem 6:

What is the output of the below program?

```
public class Problem6{  
    public static void main(String args[]) {  
        String a = "College", b = "School";  
  
        System.out.println(a+b);  
        System.out.println(b+"---"+a);  
        System.out.println(a.indexOf("oo"));  
  
        a = b;  
        b = a;  
  
        System.out.println(b.substring(2));  
        System.out.println(a == b);  
    }  
} //class
```

Problem 7:

What is the output of the below program?

```
import java.util.StringTokenizer;  
public class Problem7{  
    public static void main(String args[]) {  
        String phrase = "Jump the| SHARK, Alex";  
        StringTokenizer st = new StringTokenizer(phrase, "|, ");  
  
        System.out.println(st.countTokens());  
  
        while(st.hasMoreTokens()) {
```

```

        String temp = st.nextToken();
        if(temp.length() == 4)
            System.out.println(temp.toUpperCase());
        else
            System.out.println(temp.charAt(1));
    }
}
} //class

```

Problem 8:

What is the output of the below program?

```

import java.util.StringTokenizer;
public class Problem8 {
    public static void main(String args[]){
        String phrase = "abc-123 DEF-456+GHI|789";
        StringTokenizer st = new StringTokenizer(phrase, "- ");

        System.out.println(st.countTokens()); //line 1

        while(st.hasMoreTokens()) {
            String temp = st.nextToken();
            System.out.println(temp);
            temp = temp.substring(2);
            System.out.println(temp);
        }
    }
} //class

```

Problem 9:

What is the output of the below program?

```

public class Problem9{
    public static void main (String args[]) {
        //get the 4 words from the console
        String w1,w2,w3;

        w1 = "COFFEE";
        w2 = "coffee";
        w3 = "COffEE";

        if(w1.compareTo(w2) < 0)
            System.out.println(w1);

        if(w2.compareTo(w3) < 0)
            System.out.println(w2);

        if(w3.compareTo(w1) < 0)
            System.out.println(w3);
    }
}

```

```
    } //main  
} //class
```

Problem 10:

What is the output of the below program?

```
public class Problem10{  
    public static void main (String args[]) {  
        //get the 4 words from the console  
        String w1,w2,w3;  
  
        w1 = "COFFEE";  
        w2 = "coffee";  
        w3 = "COffEE";  
  
        if(w1.compareToIgnoreCase(w2) > 0)  
            System.out.println(w1);  
  
        if(w2.compareTo(w3) > 0)  
            System.out.println(w2);  
  
        if(w3.compareToIgnoreCase(w1) == 0)  
            System.out.println(w3);  
  
    } //main  
} //class
```

CHAPTER 7

Looping

This chapter covers the basics of repetition and looping. Showing the classic examples of printing shapes of stars, infinite loops, and empty loops.

TOPICS

1. <i>For loops</i>	115
2. <i>Nested loops</i>	122
3. <i>Break & Continue</i>	128
4. <i>Empty Loops</i>	131
5. <i>Infinite Loops</i>	132
6. <i>Exercises</i>	133

In most programs, a **loop** can be used to do something \times number of times. For the purpose of code saving it certainly helps and it can also prove quite useful.

A loop is used for repetition and contains some decision-making. The decision you need to make is what condition(s) will continue to make this loop run.

We have already seen a type of loop earlier in this book. While statements are types of loops! They will perform a set of actions X number of times, so long as the while statement logic returns true. But now let's see other types of loops in Java.

FOR LOOPS

Taking this literally, for a certain number of times, perform an action(s). At minimum, a for loop will be defined as:

```
for( loop counter; condition(s); counter changes ) {
    //code for the for
}
```

Let's break down what each section of the for loop means. Note that there are 3 sections, each of which (except the last) is separated by a semicolon.

First, the loop counter. The loop counter is a variable and is used to keep track of how many times you want the loop to run. It is always a numeric value such as an int, double, or float.

Second, the condition(s) of the loop. This is where logic comes into play. Say you wanted the loop to run 30 times. The counter would have logic that kept track of that ($counter < 30$).

Finally, the counter changes. Without this, you would have AN INFINITE LOOP! Those are quite a pain in the neck because the loop WILL NEVER END! Generally, you will either increment or decrement a loop counter, but not always. Sometimes, a program calls for a loop to have a change by 2, 3, 5, etc. Just use the correct operations.

EXAMPLE 1: A *Simple Loop*

Below is a program that will print out the current value of the loop counter. It will ask a user to enter a number (between 0 and 20), which will be how many times you want to let the loop run. This is a good way to view how a loop will run.

```
import java.util.Scanner;
public class Example1{
    public static void main(String args[]) {
        int times = 0;

        //setup the Scanner for input
        Scanner s = new Scanner(System.in);

        //get integer from user
        System.out.println("How many times? ");
        times = s.nextInt();

        if(times < 0 || times >= 20)
            System.exit(1);

        //print out the numbers from 1 to times
        for(int i = 1; i <= times; i++) {
            System.out.println(i);
        }
    } //main
} //class
```

A sample run of the above program is:

```
How many times? 6
1
2
3
4
5
6
```

The program simply prints out each number on a separate line.

EXAMPLE 2: *Sum of the Numbers from 1 to N*

Below is a program that will ask the user for a positive number bigger than 1, prompting them for an entry again until correct, and calculate the sum of the numbers from 1 to the number the user entered.

```
import java.util.Scanner;
public class Example2{
    public static void main(String args[]){
        //Scanner variable
        Scanner s = new Scanner(System.in);

        //int variables
        int sum = 1, entry = 0;

        //get entry from the user
        System.out.print("Please enter a number > 1: ");
        entry = s.nextInt();

        //while the user didn't follow directions
        while (entry <= 1) {
            System.out.print("Try again: ");
            entry = s.nextInt();
        }

        //have correct entry so find the sum
        for(int i = 2; i <= entry; i++) {
            sum += i;
        }

        //output sum
        System.out.println("The sum from 1 to " + entry +
                           " is: " + sum);
    } //main
} //class
```

A sample run of the above program is:

```
Please enter a number > 1: 0
Try again: -2
Try again: 7
The sum from 1 to 7 is: 28
```

And it proves correct because the sum of $1 + 2 + 3 + 4 + 5 + 6 + 7 = 28$.

Notice the loop counter begins at 2 in the above example, and the sum variable is initialized to 1. This saves a step in the loop, since we requested a number greater than 1 from the user.

EXAMPLE 3: *Divisible by 5*

Below is a program that will ask the user for a positive number bigger than or equal to 5, prompting them for an entry again, until correct, and displays the numbers from 1 to N that are evenly divisible by 5.

```
import java.util.Scanner;
public class Example3{
    public static void main(String args[]){
        //Scanner variable
        Scanner s = new Scanner(System.in);

        //int variables
        int sum = 1, entry = 0;

        //get entry from the user
        System.out.print("Please enter a number >= 1: ");
        entry = s.nextInt();

        //while the user didn't follow directions
        while (entry < 1) {
            System.out.println("Try again: ");
            entry = s.nextInt();
        }

        //have correct entry so find the sum
        for(int i = 1; i <= entry; i++) {
            if(i % 5 == 0)
                System.out.print(i + " ");
        }
    } //main
} //class
```

A sample run of the above program is:

```
Please enter a number >= 1: 29
5 10 15 20 25
```

Or this:

Please enter a number ≥ 1 : 0

Try again: 0

Try again: 0

Try again: 15

5 10 15

EXAMPLE 4: *Finding the Average*

Below is a program that will ask the user for a positive number bigger than or equal to 1, prompting them for an entry again until they correctly enter it, and determine the average of the numbers from 1 to N.

```
import java.util.Scanner;
public class Example4{
    public static void main(String args[]){
        //Scanner variable
        Scanner s = new Scanner(System.in);

        //double precision variables
        double avg = 0.0, entry = 0.0;

        //get entry from the user
        System.out.print("Please enter a number > 1: ");
        entry = s.nextDouble();

        //while the user didn't follow directions
        while (entry <= 1.0) {
            System.out.print("Try again: ");
            entry = s.nextDouble();
        }

        //find the sum of the numbers first
        for (double i = 1.0; i <= entry; i += 1.0) {
            avg += i;
        }

        //final value calculated here
        avg = avg / entry;

        //output average
        System.out.println("The average from 1.0 to " + entry +
                           " is: " + avg);
    } //main
} //class
```

A sample run of the above program is:

```
Please enter a number > 1: 8  
The average from 1.0 to 8.0 is 4.5
```

Notice that the loop deals with a double precision variable. Perfectly legal and needed for what we are looking for.

EXAMPLE 5: *Be Careful of the Count!*

Below is an example that will demonstrate the use of loop counters when using the counter variables outside any loops.

```
public class Example5{  
    public static void main(String args[]){  
        int i = 1;  
        int mystery = 0;  
  
        for(i = 1; i < 9; i++) { mystery++; }  
  
        System.out.println("i= " + i);  
  
        for(i = i; i > 0; i--) { mystery++; }  
  
        System.out.println("i= " + i);  
        System.out.println("mystery= " + mystery);  
    } //main  
} //class
```

The output from running the above program is:

```
i= 9  
i= 0  
mystery= 17
```

Let's explore why. Firstly, the counter variable *i* is declared at the beginning of the program. It, therefore, can be accessed anywhere below where it is declared (which in this case is the rest of the program). Keep that in mind as we continue. We then declare a *mystery* variable that will come into play later.

The first for loop sets the counter to 1 and increments the *mystery* variable each time the loop runs. We then want to print the value of *i*, which in this case, is 9. Why is it 9? Because the loop logic is checked AFTER you change the counter. We do, in fact, change the counter from 8 to 9 and then check the logic. In this case, 9 is not strictly greater than 9, so the logic becomes false and the loop ends.

But, since we have the counter outside the loop, we are able to use it again for the next one. It sets the loop counter *i* to the current value of *i* (itself). This is perfectly valid code, but one that has no effect on the program. We then run the loop -- as long as *i* is greater than 0. Just like the above, the variable is decremented before the logic is checked for the last time, hence why the output for the next value of *i* is 0.

Lastly, we print out the value of the *mystery* variable, which we can determine to be the number of times the loops ran (the overall counter). Notice it did not run 19 times, which is a common mistake one would make. Even though the logic is checked again after the variable changes, the body of the loop does not run, since the check proves false in both cases.

Now what if we modified the example to this:

```
public class Example5{
    public static void main(String args[]) {
        int i = 1;
        int mystery = 0;

        for(i = 1; i < 9; i++) {
            mystery++;
        }
        System.out.println("i= " + i);

        for(int j = i; i > 0; i--) {
            mystery++;
        }
        System.out.println("i= " + i);
        System.out.println("j= " + j);
        System.out.println("mystery= " + mystery);
    } //main
} //class
```

What do you think the output would be?

It would be nothing! The program won't even compile. Why? Because of the newly-added `println()` for the `j` variable. Here, this is a local loop counter. Once the loop has completed its run, the variable goes away and can no longer be used later in the program. Eclipse will not allow this program to even compile. The error reads "j cannot be resolved to a variable."

NESTED LOOPS

When dealing with loops, it is possible to have loops inside of one another, and another, and another, etc. These are called **nested loops**. The rule about this is that the **inner most** loop will be used first, then the one above that, and the one above that one, etc... It could be something like this, most generally:

```
for( loop counter 1; condition(s) 1; counter 1 changes ){  
    //code for the 1st for loop  
    for( loop counter 2; condition(s) 2; counter 2 changes ){  
        //code for the 2nd for loop  
        for( loop counter 3; condition(s) 3; counter 3 changes ){  
            //code for the 3rd for loop  
        }  
    }  
}
```

So, here is a short example, tracing the values of each variable along the way. Observe this code snippet:

```
for(int a = 0; a < 3; a++){  
    for(int b = 0; b < 5; b++){  
        System.out.println(a + " " + b);  
    }  
}
```

Here, the output of the above snippet would be the following:

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

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

Why is this? As mentioned, the inner-most loop would execute fully, then move to the next iteration of the above loop, and execute the inner loop again. The output reflects the values of the loop counters a and b .

Let's see further examples of nested loops.

EXAMPLE 6: *Triangle of Stars*

This program will print out a triangle of stars using two for loops, one nested inside the other. It will ask a user for a positive input less than or equal to 10.

```
import java.util.Scanner;  
public class Example6{  
    public static void main(String args[]){  
        int size = 0;  
        Scanner s = new Scanner(System.in);  
  
        //get integer from the user  
        System.out.println("Please enter the size: ");  
        size = s.nextInt();  
  
        if(size <= 0 || size > 10)  
            System.exit(1);  
  
        //loop through the program to print the triangle  
        for(int r = 1; r <= size; r++) {  
            for(int c = 1; c <= r; c++) {  
                System.out.print("*");  
            }  
            System.out.println();  
        }  
    }  
}
```

```
    } //main
} //class
```

When the user enters a correct number, a triangle is printed that will go from the top left to the bottom right. Let's see what this program does with the nested loop.

The “row” loop is first, followed by the “column” loop. Here, the row loop will have a value of 1 to begin, and the column loop will also have a value of 1. But each time the column loop changes, the value of the row DOES NOT change, until the column loop is completely finished.

Say that the user enters the number 6. Here is what this triangle would look like:

```
*
```



```
**
```



```
***
```



```
****
```



```
*****
```



```
******
```

EXAMPLE 7: *Box of Stars*

Here is another example showing how to make a hollow box of stars:

```
import java.util.Scanner;
public class Example7{
    public static void main(String args[]){
        int size = 0;
        Scanner s = new Scanner(System.in);

        //get the integer from the user
        System.out.println("Please enter the size: ");
        size = s.nextInt();

        for(int r = 1; r <= size; r++){
            for(int c = 1; c <= size; c++){
                if(c == 1 || c == size
                || r == 1 || r == size){
                    System.out.print("*");
                }else{
                    System.out.print(" ");
                }
            }
            System.out.println();
        }
    }
}
```

```
        }
    } //main
} //class
```

This program first asks the user for the size of the box. There is no error checking here, but you should add one to prevent the box from being too big.

Then, in order for you to print the box correctly, you need to make sure that you are either in the top or bottom row in addition to the left or the right column. So, if the column variable *c* equals 1 or *s*, you are in the left or right most column, so print a star. Similarly, if the row variable *r* equals 1 or *s*, you are in the top or bottom row, so print a star. Otherwise, for any other condition, print a space to keep the order of the box.

Here is an output, if the user enters the number 5 for the variable *s*:

```
*****
*   *
*   *
*   *
*****
```

EXAMPLE 8: *Pyramid of Stars*

Here is an example showing how to make a pyramid of stars:

```
public class Example8{
    public static void main(String args[]){
        int r = 0, c = 0, s = 0;

        //pyramid of stars:
        for(r=1; r<=5; r++){
            System.out.println();

            for(s=1; s <= 5-r; s++) System.out.print(" ");

            for(c=1; c <= 2*r-1; c++) System.out.print("*");
        }
    } //main
} //class
```

This program seems more complex, but it is just a matter of understanding how many spaces to print before you print a star.

Firstly, the outer loop is used for the rows. Here, there will be a set number of 5 rows of stars forming a pyramid.

The second loop s will be used to print the number of spaces before printing the number of stars. This loop goes $5-r$ times. At the first run, it will go 4 times, and print 4 spaces. The second time it will go 3 times and print 3 spaces, etc...

The third loop c will be used to print the number of stars. In order for you to keep the symmetrical shape of the pyramid, you need to have the loop run $2*r-1$ times. The first time will run 1 time (since this is the root of the pyramid). The second time will run 3 times, the third 5 times, etc...

Here is the output of the program:

```
*  
* * *  
* * * *  
* * * * *  
* * * * * *
```

You can certainly modify this program and allow the user to enter the r value. Feel free to experiment with any of the above programs. It will further develop your understanding of nested looping.

It is also possible to have multiple variables while working with a for loop. Look below:

```
for(int i = 0, k = 9; i <= k; i++, k--)  
    System.out.print(i + " ");  
}
```

This will initialize both i and k , as well as change both i and k in the counter changes. This is acceptable Java code.

This loop outputs "0 1 2 3 4 " when run.

EXAMPLE 9: *Powers of N*

The program below will calculate the sum of the powers of n to the n. Here is what that would look like:

$$1^1 + 2^2 + 3^3 + 4^4 + 5^5 + \dots$$

The trick of this program is to NOT make use of any of the Math library methods that are provided in Java.

```
public class Example9{
    public static void main(String args[]){
        int sum = 0, t = 1, n = 5;

        for(int i = 1; i <= n; i++){
            for(int j = 1; j <= i; j++){
                //will correctly multiply the i value
                //(the current n), n times as needed.
                t *= i;
            }
            sum += t; //add the desired value
            t = 1; //reset t
        }
        System.out.println("The sum is: " + sum);
    } //main
} //class
```

A sample run of this program is:

The sum is: 3413

CAREFUL: If you make n too big, or modify the program to allow the user to enter a value, you will overflow the number. This sum gets large very fast.

EXAMPLE 10: *Character Methods*

This program will get a command line argument from the user, and count the number of digits, letters, uppercase characters, and lowercase characters. This is a simple example showing some of the methods from the Character wrapper class.

```

public class Example10{
    public static void main(String args[]){
        String s = args[0];
        int upper = 0, lower = 0, letter = 0, digit = 0, space = 0;

        //loop through the String to count the different
        //types of features in the String
        for(int i = 0; i < s.length(); i++){
            if(Character.isUpperCase(s.charAt(i))) upper++;
            if(Character.isLowerCase(s.charAt(i))) lower++;
            if(Character.isDigit(s.charAt(i))) digit++;
            if(Character.isLetter(s.charAt(i))) letter++;
            if(Character.isWhitespace(s.charAt(i))) space++;
        }

        //display the result
        System.out.println("There are:\n" +
                            upper + " uppercase characters\n" +
                            lower + " lowercase characters\n" +
                            digit + " digit characters\n" +
                            letter + " letters\n" +
                            space + " white spaces.");
    } //main
} //class

```

The output from the above when the command line argument is "Today is July 22nd 2013" (entered with the quote marks):

There are:

2 uppercase characters
 11 lowercase characters
 6 digit characters
 13 letters
 4 white spaces.

BREAK & CONTINUE

As noted with the section on while looping in Chapter 5, there are two keywords that can either stop or continue iterations of a loop. If the keyword **break** appears, the loop in which the break statement appears is exited, and the program will proceed. If the keyword **continue** appears, all code below that keyword will NOT be executed, and the loop will proceed with the next iteration (if there is any more left).

Here is an example with **break**:

```
public class BreakExample{
    public static void main(String args[]) {

        for(int i = 0; i < 15; i++){
            if(i == 5) break;
            System.out.println("i= " + i);
        }

        System.out.println("Ending here...");
    } //main
} //class
```

With the above program, as soon as *i* is equal to 5, the loop will exit, and the program will continue as normal by printing “Ending here...” No other code in the loop will be executed. The output from the above program is:

```
i = 0
i = 1
i = 2
i = 3
i = 4
Ending here...
```

Now here is an example using **continue**:

```
public class ContinueExample{
    public static void main(String args[]) {

        for(int i = 0; i < 15; i++){
            if(i >= 5 && i <= 11) continue;
            System.out.println("i= " + i);
        }

        System.out.println("Ending here...");
    } //main
} //class
```

With this program, when i is between 5 and 11, no code is executed below the keyword, and the loop will proceed with the next iteration.

The output from running the above program is:

```
i = 0  
i = 1  
i = 2  
i = 3  
i = 4  
i = 12  
i = 13  
i = 14  
Ending here...
```

These keywords are used whenever they are appropriate to use in a program. Break is the most commonly used. Here is an example involving nested loops with **break**:

```
public class BreakExample2{  
    public static void main(String args[]){  
        for(int i = 0; i < 5; i++){  
            for(int j = 0; j < 5; j++){  
                if(j == 2) break;  
                System.out.println("i, j: " + i + ", " + j);  
            }  
            if(i == 3) break;  
        }  
        System.out.println("Ending here...");  
    } //main  
} //class
```

The output from the above program is:

```
i, j: = 0, 0  
i, j: = 0, 1  
i, j: = 1, 0  
i, j: = 1, 1  
i, j: = 2, 0  
i, j: = 2, 1  
Ending here...
```

The output is this way because of the break statements in both loops. Whenever the j variable equals 2, the inner loops breaks. The j variable will never reach the values 3 or 4 because the loop ends. Similarly for the i variable, it will never reach the value 4 specifically because it breaks once $i=3$.

EMPTY LOOPS

Similar to empty if statements, beware of the empty loops!!! In Java, a semicolon can be a complete statement, and sometimes even the best programmers get mixed up with a for loop syntax. This happens because of the empty statement.

Here is a quick look at this. Please look closely at this for loop:

```
for(int i = 0; i < 10; i++);
    System.out.println(i);
```

Seemingly, this will print out the value of i (0 to 9). However, the empty statement is seen as part of the for loop!!! That means, the loop will go all ten times doing nothing at all, and then print the number 0 (since that is the initial value of i).

The corrected loop should have this single semicolon removed. Here is the corrected loop:

```
for(int i = 0; i < 10; i++)
    System.out.println(i);
```

Here is a small program making light of this.

EXAMPLE 11: *The Empty Loop*

This program is supposed to print out the first 10 even numbers. However, the for loop used is not correct, since it uses the empty statement. Try to determine the output before looking.

```
public class Example11{
    public static void main(String args[]) {
```

```
int x = 0;
for(int i = 0; i < 10; i++, x+=2);
    System.out.println( x );

} //main
} //class
```

The output from running the above program is: 20. This is the case because the *x* value is incremented each time around the for loop.

INFINITE LOOPS

When programming, the worst thing to do is an **infinite loop**. This occurs because the loop does not have a terminating clause. Here are some classic infinite loops that sometimes get past the best programmers:

Loop 1:

```
int x = 3;
while(x % 2 == 1) {
    //will always be an odd number so therefore
    //it will NEVER STOP!
    x = x + 2;
}
```

Loop 2:

```
while(true) {
    //if there is no terminating clause or a
    //System.exit somewhere inside here, there
    //will definitely be trouble!

    //code below
}
```

Loop 3:

```
for(int i = 0; ;){
    //an infinite for loop! Again, make sure to include
    //a terminating clause or a System.exit.
}
```

Loop 4:

```
int x = 0;
do{
    x++;
    //always a positive number so we are in trouble!
}while(x > 0);
```

Those loops are just some examples of infinite loops you may run into. The best advice is to think carefully and plan ahead before running a loop!

EXERCISES

Directions: Read each of the following programs or code snippets CAREFULLY. For each Java program or code snippet, answer the questions following it to the best of your ability. Solutions on page 595.

Problem 1:

Given each for loop below, what is the output.

a)

```
for(int z = 0; z < 6; z++)
    System.out.print(z*3);
```

b)

```
for(int j = 7; j >= 3; j--)
    System.out.print(j%4+1);
```

c)

```
for(int i = 0, k = 1; i != k; i+=2, k++)
    System.out.print(i + " " + k);
```

d)

```
for(int s = 9, t = 3; s >= 3; s--, t++)
    System.out.print(9+s-t + "*");
```

e)

```
int q = 6;
for(int i = q % 3; i < q; i++)
    System.out.print(i%4-q);
```

f)

```
for(int i = 0; i < 6; i++){
    for(int j = 0; j < i; j++)
        System.out.print("*-*");
    System.out.print("\n");
}
```

```

g)
int x = -9;
for(int a = x+3; a <= 0; a++) {
    if(a == -5) System.out.print("555");
    else System.out.print("*");

h)
int i = 100, j = 10;
for(int k = i/10; k <= i; k+=10){
    System.out.println("D: " + k/10 + " R: " + (k % 10 + i))
}

l)
for(double d = 0.1; d <= 1.1; d += 0.1){
    System.out.print(d / 0.4 + " ");
}

m)
for(float fl = 0.25; fl < 3.25; fl+=0.25){
    System.out.print(fl + 4.0 - 0.25);
    if(fl != 3.0) System.out.print("**");
}

n)
int x = 5;
int p = x;

for(int i = p; i <= p*2; i++){
    System.out.println(i*p);
}

o)
String s = "Hello";
for(int i = s.length()-1; i >= 0; i--)
    System.out.print( s.charAt(i) );

p)
String s = "Hello";
for(int i = s.length()-1; i >= 0; i--)
    if(s[i] == 'l') System.out.print( "-" );
    else System.out.print( s.charAt(i) );

q)
String s = "Sunshine";
for(int i = 0; i <= 10; i += 2)
    System.out.print( s.charAt(i % 4) );

r)
String s = "Sunshine";
for(int i = 0; i <= 10; i += 2)
    if(i == 3 || i == 5) continue;

```

```
else System.out.print( s.charAt(i % 2) );
```

Problem 2:

Write a for loop to perform each of the following tasks:

- Print out the first 25 odd numbers.
- Print out the '^' character 10 times.
- Find the sum of the first 100 even numbers.
- Print out the first 20 numbers that end with a 5.
- Print out 5 rows of the '#' character. Each row contains 10 characters.
- Print out a table of 3 rows & 2 columns, where the numbers printed are the product of the row & column counters (begin at 1 for the value of the counters).

Problem 3:

Given each nested loop below, what is the output? If the loop is infinite, state so.

```
a)
for(int i = 0; i < 10; i+=2) {
    do{
        System.out.print( ++i + " ");
        if(i % 2 == 1)
            System.out.println("odd");
    }while(i < 6);
}

b)
for(int i = 7; i >= 0; i-=3) {
    do{
        System.out.print(i + " ");
    }while(i > 4);
}

c)
int x = 0;
while(x+3 < 10) {
    ++
    System.out.println(x--);
}

d)
int q = 7;
while(q != 3) {
    for(int i = q; i >= 1; i--) {
        System.out.print(q-i + " ");
    }
    q--;
}
```

```

e)
for(int x = 10; x >= 0; x--) {
    if(x % 3 == 0) continue;
    else{
        do{
            System.out.print(x + " ");
            x--;
        }while(x > 5);
    }
}

f)
for(int i = 1; i <= 5; i++) {
    for(int j = 1; j <= i; j++) {
        for(int k = 0; k < j; k++)
            System.out.print(j);
        System.out.println();
    }
}

g)
int x = 8;
while(x < 10) {
    while(y < 3) {
        while(y < x) {
            System.out.print("+");
            y++;
        }
        System.out.println();
    }
    x++;
}

```

Problem 4:

Write a program to perform the following tasks:

1. The user enters a command line argument for a positive integer n .
2. If the entered number is not positive, it will exit the program.
3. The program than prints a triangle of numbers looking like this:

```

3
32
321

```

Problem 5:

Write a program to perform the following tasks:

1. The user enters a command line argument for a positive integer n .
2. If the entered number is not positive, it will exit the program.
3. The program than prints something looking like this:

```
4  
34  
234  
1234
```

Problem 6:

Write a program to perform the following tasks:

1. The user enters a command line argument for an odd positive integer n.
2. If the entered number is not an odd positive, the program will exit.
3. The program than prints a square looking like this:

```
*****  
* * * *  
*****  
* * * *  
*****  
* * * *  
*****
```

Problem 7:

Write a program to perform the following tasks:

1. The user enters a command line argument for a positive odd integer n.
2. If it is not positive, the program will exit.
3. The program prints an "X" to the screen looking like this:

```
X   X  
X X  
 X  
X X  
X   X
```

Problem 8:

Write a program to perform the following tasks:

1. The user enters a command line argument for a positive odd integer n.
2. If it is not positive or odd, the program will exit.
3. The program prints a "Z" to the screen looking like this:

```
XXXXX  
 X  
 X  
 X  
XXXXX
```

Output Tracing

Problem 9:

```
public class Problem9{
    public static void main(String args[]) {
        int a = 40;
        int q = a-3;

        do{
            a -= 3;
            System.out.println(a-q);
        }while(a >= 5);

        System.out.println("DONE");
    } //main
} //class
```

1. What is the output from running the above program?

Problem 10:

```
public class Problem10{
    public static void main(String args[]) {
        int x = 0;
        final String word = "Dumb!";

        x = Integer.parseInt(args[0]);

        for(int i = 0; i < x; i++);
            System.out.println(word);

        if(x % 2 == 0);
            System.out.println("Even!");
    } //main
} //class
```

1. What is the output from the above program when the argument is 5?

Problem 11:

```
public class Problem11{
    public static void main(String args[]) {
        int x = 0;
        final String word = "Dumb!";

        x = Integer.parseInt(args[0]);

        int i = 0;
        while(i < x) i++;
        System.out.println(word + " " + x);
    }
}
```

```

        if(x % 2 == 1);
            System.out.println("Odd!");
    } //main
} //class

```

1. What is the output from the above program when the argument is 5?

Problem 12:

```

public class Problem12{
    public static void main(String args[]){
        int x = 0;
        final String word = "Dumb";

        x = Integer.parseInt(args[0]);

        for(int i = 0; i < x; i++)
            for(int j = 0; j < x; j++)
                System.out.println(word + " " + " " + (word+"er"));

        if(x+1 % 9 == 3)
            if(x - 3 == 0);
                System.out.println("Zero is the hero!");
    } //main
} //class

```

1. What is the output from the above program when the argument is 2?

Problem 13:

```

public class Problem13{
    public static void main(String args[]){
        int x = 4;
        if(x % 2 == 1)
            System.exit(1);

        do{
            System.out.println(y(x, 'A'));
            x++;
        }while(x < 10);
    }

    public static int y(int a, char c){
        return (int)c - a;
    }
}

```

1. What is the output of the above program?

Problem 14:

```
public class Problem14{
    public static void main(String args[]){
        short x=0, y=0;
        x = Short.parseShort(args[0]);
        y = Short.parseShort(args[1]);

        if((y+x)%3 == 0){
            System.out.println("Whoops!");
            System.exit(1);
        }

        do{
            while(x <= y){
                System.out.println(x+y);
                x++;
            }
        }while(++y >= 15);
    }
}
```

1. What is the output of the above program when:
 - a. args[0] = 6 and args[1] = 9?
 - b. args[0] = 6 and args[1] = 10?
 - c. args[0] = 10 and args[1] = 6?

Problem 15:

```
public class Problem15{
    public static void main (String [] args){
        int x = 3;
        do{
            do{
                if(++x % 2 == 0) break;
            }while(x % 2 == 1);
            System.out.println(x);
        }while(x < 10);
    } //main
}
```

1. What is the output of the above program?

Additional Problems

DIRECTIONS: Answer each problem below to the best of your ability. Solutions are not provided in this book.

Problem 16:

Write a program to perform the following tasks:

1. The user enters a command line argument for a positive integer n.
2. If it is not positive, the program will exit.
3. The program prints a cross to the screen looking like this:

```
*  
*  
*****  
*  
*
```

Problem 17:

Write a program to perform the following tasks:

1. The user enters a command line argument for a positive odd integer n.
2. If it is not positive, the program will exit.
3. The program prints an diamond to the screen looking like this:

```
*  
* *  
*   *  
* *  
*
```

Problem 18:

Write a program to perform the following tasks:

1. Ask the user for a value of a double precision value N greater than or equal to 1.
2. The program will calculate and display the answer to the following series:

$$1.0/1.0 + 1.0/2.0 + 1.0/3.0 + \dots + 1.0/N$$

Problem 19:

Write a program to calculate the value of *pi* from the infinite series listed below. Since the program needs to terminate at some point to see the value, have your loop run to (and including) 5,000,000.0

$$4 - 4.0/3.0 + 4.0/5.0 - 4.0/7.0 + 4.0/9.0 \dots$$

Problem 20:

Write a program to perform the following tasks:

1. The user enters a command line argument for their name followed by their age.
2. If the age is less than 5 or bigger than 100, the program exits.

3. The program will print the name entered, age times, meaning if the user entered 10 for the age, their name will print 10 times.

Problem 21:

Write a program to perform the following tasks:

1. The user enters a command line argument for their first name.
2. The program will print every other character in the name entered.

Problem 22:

Write a program to perform the following tasks:

1. The user enters a command line argument for a word of an odd length.
2. If the word entered is not of an odd size, the program will repeatedly ask the user for another word of an odd length.
3. The program will print the first and last character of the word n times where n is the size of the string.

Problem 23:

Write a program to perform the following tasks:

1. The user enters a command line argument for a word beginning with a vowel.
2. If the word does not begin with a vowel, the program will exit.
3. The program will print only the consonants of the word.

Problem 24:

Write a program to perform the following tasks:

1. The user enters a command line argument for a word.
2. The program will print the word in the following pattern:

If the word is "hello":

h
he
hel
hell
hello

CHAPTER 8

Methods

This chapter discusses the various types of methods that can be written in Java. It will show the passing by value, by reference, declaration of methods, and method parameters.

TOPICS

1. <i>Method Basics</i>	144
2. <i>Calling Methods</i>	145
3. <i>Types of Methods</i>	145
4. <i>Parameters</i>	146
5. <i>Overloading Methods</i>	152
6. <i>Exercises</i>	157

In Java, a **method** is a block of code that performs a certain task. This is ***almost*** the same as a C++ function. There are some small differences which we will explore.

A method can be **private**, **protected**, or **public**, as well as **static** or **non-static**. The return types can be anything primitive (int, float, etc...), an Object, or user-defined.

METHOD BASICS

Here is how to define a Java method:

```
identifier type Name( argument(s) ) {  
    //code here  
}
```

Where in the above, *identifier* is either public, private, or protected; *type* is the return type of the method; *Name* is a useful name of the method, and *argument(s)* are the parameters to pass to the method (if any).

The *type* is the specific data type that will be returned by the method. Any of the primitive data types can be used with methods (int, float, Boolean, etc...) in addition to any user-defined types (classes and objects). However, there is a special type listed below:

void

Does not return a value. Mostly used for printing information.

A simple method void method is defined below:

```
private static void printMessage(){  
    System.out.println("Hello and welcome! ");  
}
```

This method would print out “Hello and welcome!” on screen when used. But how do you use a method?

CALLING METHODS

A method is used when it is called. In order to call a method, all you need to do is type the EXACT name of the method with any appropriate parameters in order to be called.

Below is a short program that uses the above void method:

EXAMPLE 1: Void Method

This program simply outputs a message to the console via a void method.

```
public class Example1{
    public static void main(String args[]) {
        printMessage();
    } //main

    static void printMessage() {
        System.out.println("Hello and welcome!");
    }
} //class
```

TYPES OF METHODS

A method that uses instance variables of that class is called an instance method. This is the default for a method.

A method that uses **NO instance variables** can be declared static. It must NOT use any of them, otherwise there will be a compiler error. The static methods tend to compute something from arguments of that method instead of using the variables.

Here is an example of a static method:

```
public static double mean(int n1, int n2, int n3) {
    int sum = 0;
    double ans = 0.0;

    sum += n1 + n2 + n3;
    ans = sum / 3.0;

    return ans;
```

```
} //method mean
```

And another:

```
private static long twice(long n){  
    return 2*n;  
} //method twice
```

And another:

```
public static String message(String name, int age){  
    String str;  
  
    str = "Hello " + name + "! You are " + age + " years old! ";  
  
    return str;  
} //method message
```

Etc, etc, etc...

PARAMETERS

A parameter (or argument, as we know already), is a piece of data given to a method. A parameter to a method is also called a formal parameter.

PLEASE NOTE: all parameters that are not of object types are PASSED BY VALUE! All others are PASSED BY REFERENCE (including arrays).

Let's see a simple class and a simple instance method:

EXAMPLE 2: An Instance Method

```
public class Example2{  
    private static int s=3, t=7;  
  
    public static void main(String args[]){  
        System.out.println("sum: " + sum(s,t));  
    }  
}
```

```

} //main

private static int sum(int a, int b) {
    return a+b;
} //sum
} //class

```

The above program will compute the sum of the 2 instance variables. The reason the method is static is because the instance variables are static, as well as the fact that the method is being called from the static main() method.

Now let's see an example with a String.

EXAMPLE 3: *Object example*

```

public class Example3{
    static String word1 = "Cats";
    static String word2 = "Dogs";

    public static void main(String args[]){
        System.out.println("Before call:\nword1: " +
                           word1 + "\nword2: " + word2);

        //call the switches() method
        switches(word1,word2);

        System.out.println("\nAfter call:\nword1: " +
                           word1 + "\nword2: " + word2);
    }

    private static void switches(String w1, String w2){
        w1 = w2;
        w2 = "Changed!";
    } //method
} //class

```

Here, we wrote a method called switches that takes two String arguments. It is seen that we are changing the values of the words in the method. The words in the main method DO NOT change. Once we attempt to change a variable inside a method, we lose the reference to it from the place we called it.

Here is the output from running the above program:

Before call:

word1: Cats

word2: Dogs

After call:

word1: Cats

word2: Dogs

EXAMPLE 4: *Printing Characters*

This example makes use of a method that prints a certain character n times.

```
import java.util.Scanner;
public class Example4{

    private static void multiPrint(String c, int x){
        //print the character x times
        for(int i = 0; i < x; i++)
            System.out.print(c);
        System.out.println();
    }

    public static void main(String args[]){
        int n = 0;
        String c = "";

        System.out.println("Please enter a number and character:");

        //Scanner for input of both the integer and character
        Scanner s = new Scanner(System.in);
        n = s.nextInt();
        c = s.next();

        //call method to print that character
        multiPrint(c, n);
    } //main
} //class
```

The method `multiPrint()` is called in the `main()` method after the user enters the integer and the character to print. The method is void because it is simply used for printing.

A sample run of the program above may be:

Please enter a number and a character: 5 ^
^^^^^

EXAMPLE 5: *Sum of 1 to N*

This program will show a method that calculates the sum of the numbers from 1 to n as specified by the user. The method takes 1 argument, which is the n . The method also returns an integer that is the sum of the numbers.

```
import java.util.Scanner;
public class Example5{
    private static int sumOfNumbers(int x){
        int sum = 0;

        //find sum by looping
        for(int i = 1; i <= x; i++)
            sum += i;

        //return the calculated value
        return sum;
    }

    public static void main(String args[]){
        int n = 0;

        System.out.println("Please enter a number for the sum:");
        Scanner s = new Scanner(System.in);
        n = s.nextInt();

        System.out.println("The sum is: " + sumOfNumbers(n));
    } //main
} //class
```

A sample run of this program may be:

```
Please enter a number to find the sum: 100
The sum is 5050
```

EXAMPLE 6: *Counting Digits*

This program will count the digits of an integer with the use of a method called `countNines()`. Here, the method will count the nines in an integer argument, but can

certainly be modified to count any digit, but here, it will count the nines. It will return an integer, which represents the total.

```
import java.util.Scanner;
public class Example6{
    private static int countNines(int n) {
        int count = 0;
        while(n > 0){
            //found a 9 so increment count variable
            if(n % 10 == 9) count++;
            n = n/10; //truncate the number
        }

        //return calculated value
        return count;
    }
    public static void main(String args[]){
        int n = 0;

        //get the integer from the user
        System.out.print("Please enter a positive integer: ");
        Scanner s = new Scanner(System.in);
        n = s.nextInt();

        //not positive so keep trying
        while(n < 0){
            System.out.print("\nNot positive, try again: ");
            n = nextInt();
        }

        //call method and display result
        System.out.println("\nThere are " + countNines(n)
                        + " nines in the number");
    } //main
} //class
```

The method itself will take the integer and look at the last digit. If it is a 9, it increments the count variable (since that is what we want). The modulo by 10 will isolate the last digit of the number. The next step would be to divide the number by 10, and repeat the cycle.

A sample run of the above program may be:

```
Please enter a positive integer: -3
Not positive, try again: -1
```

Not positive, try again: 14599293

There are 3 nines in the number

EXAMPLE 7: *Methods to Print Stars*

The below program prompts the user for a number larger than 0. If it is not entered correctly, the program terminates. Otherwise, it calls on two methods, both of which will print stars in various patterns.

```
import java.util.Scanner;
public class Example7{
    public static void main(String args[]) {
        //Scanner
        Scanner s = new Scanner(System.in);
        int size = 0;

        System.out.println("Enter an integer > 0: ");
        size = s.nextInt();

        if(size < 1) System.exit(1);

        //call the method for printing
        printTriangle(size);

        System.out.println();

        //call the method for printing
        printSquare(size);
    } //main

    //method for printing a triangle of stars
    private static void printTriangle(int size) {
        //loop through to print the triangle
        for(int r = 1; r <= size; r++) {
            for(int c = 1; c <= r; c++) {
                System.out.print("*");
            }
            System.out.println();
        }
    } //printTriangle()

    //method for printing a box of stars
    private static void printSquare(int size) {
        //loop through to print the square
        for(int r = 1; r <= size; r++){
            for(int c = 1; c <= size; c++) {
                if(c == 1 || c == size)
```

```

        || r == 1 || r == size){
                System.out.print("*");
        }else{
                System.out.print(" ");
        }
}
System.out.println();
}
} //printSquare()
} //class

```

The program above uses code, as seen in Chapter 7 on loops.

A sample run of the above program can be:

Enter a number > 0: 5

```

*
**
***
****
*****
*****
*   *
*   *
*   *
*   *
*****

```

OVERLOADING METHODS

In Java, many methods are already written for you, such as the equals() method in the String class. If you wish to create your own version of a method that may take different data types as parameters, you can do so via something called **overloading**.

Say we have a short program as follows:

```

public class OverloadExample{
    public static void main(String args[]){
        talk("Hey there!");
        talk("Hey there my name is John", "John");
    } //main
    public static void talk(String t){
        System.out.println(t);
    }
}

```

```

        public static void talk(String t, String name){
            System.out.println(name + " said this: " + t);
        }
    } //class
}

```

Here, we see that there are two methods called talk, both of which return void, and are named the same. The only difference is the arguments of these methods. This is allowed in Java. We have now overloaded the talk() method.

Another example may be as follows. Say we wanted to find the average of numbers in a program. We can write a method called avg() that will return the average accordingly. We also want there to be a method to handle 2, 3, and 4 numbers. We can do that with overloaded methods:

```

public class Overload2{
    //method to find average of 2 numbers
    private static double avg(int n1, int n2){
        return( (n1+n2)/2.0 );
    }

    //method to find average of 3 numbers
    private static double avg(int n1, int n2, int n3){
        return( (n1+n2+n3)/3.0 );
    }

    //method to find average of 4 numbers
    private static double avg(int n1, int n2, int n3, int n4){
        return( (n1+n2+n3+n4)/4.0 );
    }

    public static void main(String args[]){
        //version with 2 arguments
        System.out.println("The avg of 3 and 5 is: "
            + avg(3,5) );

        //version with 3 arguments
        System.out.println( "The avg of 3, 4 and 9 is: "
            + avg(3,4,9) );

        //version with 4 arguments
        System.out.println( "The avg of 4, 8, 10 and 8 is: "
            + avg(4,8,10,8) );
    } //main
} //class

```

It is perfectly fine to have 3 different avg() methods in the program above.

BE CAREFUL: It is NOT okay to have two methods with the same name and DIFFERENT return types. This will produce an error in Java. Specifically, it will say “Duplicate method [name(type)] in [class name].”

EXAMPLE 8: Arithmetic Showcase with Methods

The below program prompts the user for 3 numbers. It will then find the sum, difference, product, and quotient for these numbers. Please pay attention to the order of the methods called upon, as well as the parameters for each.

```
import java.util.Scanner;
public class Example8{
    public static void main(String args[]) {
        //Scanner
        Scanner s = new Scanner(System.in);
        int n1, n2, n3;

        //initialize variables
        n1 = n2 = n3 = 0;

        //prompt user and obtain entries
        System.out.println("Enter 3 integers: ");
        n1 = s.nextInt();
        n2 = s.nextInt();
        n3 = s.nextInt();

        //print out the values of the method returns
        System.out.println( sum(n1, n2) );
        System.out.println( diff(n1, n2) );
        System.out.println( prod(n1, n2) );
        System.out.println( sum(n1, n2, n3) );
        System.out.println( diff(n1, n2, n3) );
        System.out.println( quot(n1, n2) );
        System.out.println( quot(n2, n3) );

        //call the mystery methods
        System.out.println( mystery(n2, n3) );
        System.out.println( mystery(n1, n2, n3) );
    } //main

    //method to return the sum of two int variables
    private static int sum(int a, int b) {
        return a + b;
    }
}
```

```

//method to return the sum of three int variables
private static int sum(int a, int b, int c) {
    return a + b + c;
}

//method to return the difference of two int variables
private static int diff(int a, int b) {
    return a - b;
}

//method to return the difference of three int variables
private static int diff(int a, int b, int c) {
    return a - b - c;
}

//method to return the product of two int variables
private static int prod(int a, int b) {
    return a * b;
}

//method to return the product of three int variables
private static int prod(int a, int b, int c) {
    return a * b * c;
}

//method to return the quotient of two int variables
private static int quot(int a, int b) {
    if(b == 0) return -1;
    return a / b;
}

//mystery method 1
private static int mystery(int a, int b) {
    if(a < b) {
        return sum(a, b) + prod(a, b);
    }else {
        return diff(a, b) + prod(a, b);
    }
}

//mystery method 2
private static int mystery(int a, int b, int c) {
    if(a < c) {
        return sum(a, b) + prod(b, c);
    }else {
        return diff(c, b) + prod(a, b);
    }
}
} //class

```

The program utilizes overloaded methods for sum, diff, and prod. They are named the same, with the same return type, but take different amounts of parameters.

A sample run of the above program can be:

Enter 3 integers:

7 2 5

9

5

14

14

0

3

0

17

17

Let's observe why this is. We know, from the above entries from the user, that:

$$n1 = 7, n2 = 2, n3 = 5$$

The first method call is for *sum(n1, n2)* or *sum(7, 2)*. This returns the value of 9 and prints that value to the screen.

The next method call is for *diff(n1, n2)* or *diff(7, 2)*. This returns the value of 5 and prints that value to the screen.

The next method call is for *prod(n1, n2)* or *prod(7, 2)*. This returns the value of 14 and prints that value to the screen.

The next method call is for *sum(n1, n2, n3)* or *sum(7, 2, 5)*. This returns the value of 14 and prints that value to the screen.

The next method call is for *diff(n1, n2, n3)* or *diff(7, 2, 5)*. This returns the value of 0 and prints that value to the screen.

The next method call is for *quot(n1, n2)* or *diff(7, 2)*. This returns the value of 3 and prints that value to the screen.

The next method call is for *quot(n2, n3)* or *diff(2, 5)*. This returns the value of 0 and prints that value to the screen.

The next method call is for *mystery(n2, n3)* or *mystery(2, 5)*. Breaking down the code inside the method, it first checks if the value of the *a* parameter is less than the value of the *b* parameter. In this case, it evaluates to true, so the return value of the method would be *sum(2, 5) + prod(2, 5)*, which in this case is $7 + 10$, or 17. It prints that value to the screen.

The next method call is for *mystery(n1, n2, n3)* or *mystery(7, 2, 5)*. Breaking down the code inside the method, it first checks if the value of the *a* parameter is less than the value of the *c* parameter. In this case, it evaluates to false so the return value of the method would be *diff(5, 2) + prod(7, 2)*, which in this case is $3 + 14$, or 17. It prints that value to the screen.

This was a long example, for sure, but it is helpful to know overloaded methods and how to use them.

EXERCISES

DIRECTIONS: Answer each question below to the best of your ability. Solutions on page 601.

Problem 1:

Write **header lines** for each method description below. DO NOT write the actual method.

- a) A method called **sum** that will return the sum of 3 integer parameters.
- b) A method called **printChar** that will print a character *n* number of times.
- c) A method called **triangle** that will print a forward facing triangle of the character **'*'**. The argument is the number of rows in the triangle.
- d) A method called **perfectSq** that will print the first 25 perfect squares.
- e) A method called **isLeapYear** that will return true if the specified integer argument represents a leap year.
- f) A method called **isOdd** that will return true if the integer argument is odd.
- g) A method called **isEven** that will return true if the long argument is even.
- h) A method called **isEvenDiv** that will return true if the integer argument specified is even and divisible by 5.
- i) Write a method called **evenSum** to compute the sum of the even integers between *a* and *b* inclusive (including *a* and *b*).

- j) Write a method called **oddSum** to compute the sum of the odd integers or numbers divisible by 5 between a and b inclusive (including a and b).
- k) A method called **reverseStr** that will reverse a string argument passed to it.
- l) A method called **toLower** that will convert all uppercase characters of a string parameter to lower case and return the converted string.
- m) A method called **getSub** that will return the nth character in a string argument.
- n) A method called **inBetween** that will return the median value of two double precision numbers.
- o) A method called **isAllEven** which returns true if all digits in an integer parameter are even. It returns false otherwise.
- p) A method called **isAlpha**, which returns true if a String parameter contains all letters (a-z and A-Z) and false otherwise.
- q) A method called **isAlphaNumeric**, which returns true if a String parameter contains all letters and numbers (a-z, A-Z, 0-9) and false otherwise.
- r) A method called **dupe8** that will duplicate any occurrence of an 8 in an integer parameter and print the new value to the screen.
- s) A method called **drop8** that will drop any occurrence of an 8 in an integer parameter and return its new value.
- t) A method called **squareNum** that will print the square of each digit in an integer parameter.
- u) A method called **frac**, taking two integer parameters, that will return the value of an argument 1 divided by argument 2. If argument 2 is zero, the method returns -1.
- v) A method called **inString**, taking a String parameter and a char parameter, which returns the number of times the character parameter appears in the String parameter.
- w) A method called **stringInString**, taking two String parameters, which returns the number of times the second String parameter appears in the first String parameter.

Problem 2:

Write each method *in full* from the above section. It is implied that all appropriate libraries are being used. For example, if you need something from the java.util library, you do not need to write import java.util.*.

Problem 3:

In the world of mathematics, there are many different types of summations and infinite series. Write a Java method that will perform the following mathematical tasks:

- a) Write a method to calculate the sum of the numbers from a to b .

- b) Write a method to calculate the product of the numbers from a to b . (For this question, you can assume an integer value).
- c) Write a method to calculate an alternating series of values from a to b . The first term of the series will always be positive. An example may be the sum of $1 - 2 + 3 - 4$, etc.
- d) Write a method to calculate the sum of the first 10 fractions of the form $1/(2^n)$. This will look like the series $1 + 1/2 + 1/4 + 1/8 + \dots + 1/1024$.
- e) Write a method to calculate the sum of the first 1,000 odd numbers.

Problem 4:

Write a program that performs the following tasks:

- 1. The user enters a command line argument for an integer n .
- 2. If the number is negative or 0, the program will terminate.
- 3. The program will calculate the product of digits with the use of a method called **prodDigits** that takes an integer parameter.

Problem 5:

Write a program that performs the following tasks:

- 1. The user enters a command line argument for an integer n .
- 2. If the number is negative or 0, the program will terminate.
- 3. The program will calculate the sum of the odd digits with the use of a method called **sumOdd** that takes an integer parameter.

Problem 6:

Write a program that performs the following tasks:

- 1. The user enters a command line argument for an integer n followed by a String c .
- 2. If the number is negative or 0, the program will terminate.
- 3. The program will print the character c , n times on n lines with the use of a method **printChar** which takes an integer argument and a String argument.

Problem 7:

Write a program that performs the following tasks:

- 1. The user enters a command line argument for a String called *word*.
- 2. The program will calculate the number of vowels that are uppercase with the use of a method called **vowelCount** that takes a string parameter.

Problem 8:

Write a program that performs the following tasks:

1. The user enters a command line argument for an integer n.
2. If the number is negative, the program will terminate.
3. The program will calculate and print the binary equivalent of the number with a method called **toBinary**. The method takes an integer parameter.

Problem 9:

Write a program that performs the following tasks:

1. Ask the user for a positive 5 digit integer n.
2. If the number is not 5 digits in length, the program will terminate.
3. The program will determine if the number is a palindrome with a method called **isPalindrome**. A palindrome is a number that can be read the same forwards and backwards. 10001 is a palindrome, as well as 40204. 11210 is not a palindrome.

Problem 10:

Write a program that performs the following tasks:

1. Prompt the user to enter 3 integers (use a Scanner).
2. Print the integers to the screen with a method called **printNums** taking the 3 integers as their arguments.
3. Print the largest number of the set to the screen within that method.
4. Print the smallest number of the set to the screen within that method.

Problem 11:

Write a program that performs the following tasks:

1. Prompt the user to enter a positive, 7 digit integer.
2. If the user does not enter a correct value, the user will keep trying until the correct value is entered.
3. Write and utilize a method called **drop3**, which takes the entered number as its parameter, that will drop any 3 that appears in the parameter and return the resulting value. An example is:

`drop3(6783824) = returns 678824`

`drop3(1334544) = returns 14544`

Problem 12:

Write a program that performs the following tasks:

1. Prompt a user to enter three, 3 digit integers.
2. If any integer is not entered correctly, the program will terminate.

- With the use of a method called **combineNums**, which takes the 3 entered integers as its parameters, print a returned value from the method representing the combined values of each of its digits. An example is:

`combineNums(123, 456, 789) = returns 147258369`

`combineNums(321, 987, 456) = returns 394285176`

Problem 13:

Write a program that performs the following tasks:

- Prompt a user to enter a positive 6 digit integer (the variable for entry will be called *num*).
- If the entered value is not correct, the user enters the value again until it is correct.
- Prompt the user to enter a positive integer between 1 and 5 (the variable for entry will be called *X*).
- If the entered value is not correct, the program will terminate.
- With the use of a method called **upX**, taking two parameters reflecting *num* and *X*, print a returned value from the method that reflects the increase of each digit of *num* by *X*. If the digit increase will make a value of 10 or more, do not add that to the final result. An example is below:

`upX(523412, 3) = returns 856745`

`upX(666444, 5) = returns 999`

Problem 14:

Write a program that performs the following tasks:

- The user enters an integer value from the command line.
- With the use of a method called **scary**, taking an integer parameter reflecting the entered value, return a value of true if the parameter is evenly divisible by 13 and false otherwise.
- If the returned value to the main method is true, print “AHHH!” to a JOptionPane, otherwise print “Whew!” to the console.

Problem 15:

Write a program that performs the following tasks:

- The user enters two Strings from the command line.
- Write and utilize a method called **dupeString**, that takes two String arguments. Return a String reflecting duplicated values of the second parameter in the first

parameter. If the second parameter is not contained in the first, return the first parameter.

3. Write and utilize a method called **dropString**, that takes the same two String arguments. Return a String reflecting dropped values of the second parameter in the first parameter. If the second parameter is not contained in the first, return the first parameter. For example:

dupeString("ABCDEFG", "BCD") returns ABCDBCDEFG

dupeString("Hello world", "l") returns Hellllo worlld

dupeString("ABCDE", "Z") returns ABCDE

dropString("ABCDEFG", "BCD") returns AEFG

dropString("Hello world", "l") returns Heo word

dropString("ABCDE", "Z") returns ABCDE

Problem 16:

1. The user is prompted to enter a choice of either F, C or K (can be upper or lowercase entry), which represents Fahrenheit, Celsius or Kelvin.
2. If the entry is not correct, the user will enter a value until it is correct.
3. The program then asks the user for an integer value, representing the temperature in the given scale.
4. Write 6 methods for this program, that reflect the following temperature conversions:

F to C	$(F - 32) * 5/9$
F to K	$(F - 32) * 5/9 + 273.15$
C to F	$(C * 9/5) + 32$
C to K	$(C + 273.15)$
K to F	$(K - 273.15) * (9/5) + 32$
K to C	$(K - 273.15)$

5. Based on what the user enters, you will call the appropriate method to print the converted values to the other two temperature scales to the console. If the user entered F, you would utilize the FtoC and FtoK methods and print their values.

Output Tracing

Problem 17:

```
public class Problem17{  
    private static int series(int a, int b){
```

```

        int sum = 0;
        for(int i = a; i <= b; i++)
            sum += i;
        return sum;
    }
    private static int series2(int a, int b, boolean x){
        int sum = 0;
        if(x){
            sum = series(a, b);
        }else{
            sum = a + b;
        }
        return sum;
    }
    private static int series3(int a, int b, boolean x, boolean y){
        int sum = 0;
        if(x && y){
            sum = series(b,a);
        }else if(x){
            sum = series2(a,b,y);
            sum += b*a;
        }else{
            sum = a+b*a;
        }
        return sum;
    }
    public static void main(String args[]){
        System.out.println( series(1,10) );
        System.out.println( series2(10, 20, true) );
        System.out.println( series3(1, 20, true, false) );
    } //main
} //class

```

- What is the output of the above program?

Problem 18:

```

public class Problem18{
    public static double f(double n, double m){
        double g = m/n;
        g = g*n;
        System.out.println(g);
        return g;
    }
    public static void g(double x, double y){
        f(x,y);
        double q = f(y,x);
        double s = f(x,y);
        q = q-s;
        System.out.println(s + " -- " + q);
    }
}

```

```

public static void main(String args[]) {
    double a = 3.0;
    double b = 6.0;
    g(a,b);
} //main
} //class

```

1. What is the output of the above program?
2. If the method f was a void method, what would happen?

Problem 19:

```

public class Problem19{
    public static void func(int y, int z){
        y *= z;
        System.out.println(y + " + " + z
                           + " = " + (y+z));
    }
    public static void main(String args[]){
        int u = 9;
        for(int i = 1; i <= 4; i++) func(i, u);
    } //main
} //class

```

1. What is the output of the program?
2. If the loop logic was changed to $i \leq 8$, what is the output?

Problem 20: (variation on Problem 19)

```

public class Problem30{
    public static void func(int y, int z){
        y *= z;
        System.out.println(y + " + " + z
                           + " = " + (y+z));
    } //func
    public static void main(String args[]){
        int u = 9;
        for(int i = 1; i <= 4; i++) func(++i, u);
    } //main
} //class

```

1. What is the output of the program?
2. If the loop logic was changed to $i \leq 8$, what is the output?

Problem 21:

```
public class Problem21{
    public static int mystery3(int q){
        q = 3 - q * 2;
        return q;
    }
    public static void mystery2(int x){
        int ans = mystery3(x);
        System.out.println(ans--);
    }
    public static void mystery(int i){
        mystery2(i);
    }
    public static void main(String args[]){
        mystery(5);
    } //main
} //class
```

1. What is the output of the above program?

Problem 22:

```
public class Problem22{
    static void func(int x, int q){
        return x*q;
    }
    public static void main(String args[]){
        int x = 9, q = 3;
        x = q = func(x, q);
        System.out.println(x);
    } //main
} //class
```

1. What is wrong with the above program? Explain briefly.

Problem 23:

```
public class Problem23{
    private static int x(int y, int z){
        if(z == 0)
            return 100 * y;
        else if(z < 10 && z > 0)
            return 100000*y*z;
        else
            return 1000 + y + z;
    }
    public static void main(String args[]){
        int a = Integer.parseInt(args[0]);

        System.out.println( x(a, 0) );
        System.out.println( x(a, 10) );
    }
}
```

```

        System.out.println( x(a, 8) );
    } //main
} //class

```

1. What is the output of the program when the user enters the following values of a?
- 10
 - 100
 - 100000
 - 3

Problem 24:

```

public class Problem24{
    private static boolean search(String x, int s, int f){
        if(x.charAt(s) == x.charAt(f)){
            System.out.println("MATCH!");
            return true;
        }
        return false;
    }
    public static void main(String args[]){
        String s = "We are happy!";
        search(s, 1, 5);
        if(search(s, 2, 6))
            System.out.println("VERY HAPPY!");
        else
            System.out.println("SO SAD!");
        if(search(s, 5, 7))
            System.out.println("Jackpot!");
    } //main
} //class

```

1. What is the output from the above program?

Problem 25:

```

public class Problem25{
    static short x=0, y=0;
    public static void main(String args[]){
        x = Short.parseShort(args[0]);
        y = Short.parseShort(args[1]);
        switch(x%2){
            case 0:
                System.out.println(f());
                break;
            case 1:

```

```

        System.out.println(-1*f());
        break;
    }
}
public static short f(){
    if(x < 0 && y < 0)
        return (short)-1;
    else if(x < 0 && y >= 0)
        return (short)(4 + y);
    else if(x >= 0 && y < 0)
        return (short)(4 + x);

    return (short) (x * y);
}
}

```

1. What is the output of the above program when:
 - a. args[0] = 0 and args[1] = 3?
 - b. args[0] = 7 and args[1] = -2?
 - c. args[0] = -2 and args[1] = -2?

Problem 26:

```

public class Problem26{
    static void fix(int x, int y){
        x = y;
        y = x;
    }
    static void swaps(int x, int y){
        int t = x;
        x = y;
        y = t;
    }
    public static void main(String args[]){
        int x = 0, y = 0, i = 0;
        x = Integer.parseInt(args[0]);
        y = Integer.parseInt(args[1]);

        while(i < 5){
            swaps(x, y);
            fix(x, y);
            i++;
        }
        System.out.println(x + " " + y);
    }
}

```

1. What is the output of the above program when:
 - a. args[0] = 4 and args[1] = 4?

- b. `args[0] = 5` and `args[1] = 8?`

Problem 27:

```
public class Problem27{
    static boolean yes(int x, int y){
        if(x*y % 3 == 0) return true;
        return false;
    }
    public static void main(String args[]){
        int x = 0, y = 0;
        x = Integer.parseInt(args[0]);
        y = Integer.parseInt(args[1]);

        while(x < y){
            if(yes(x, y))
                System.out.print(x);
            else
                System.out.print(y);
            x++;
        }
    }
}
```

1. What is the output of the above program when:
 - a. `args[0] = 3` and `args[1] = 8?`
 - b. `args[0] = 8` and `args[1] = 3?`
 - c. `args[0] = 0` and `args[1] = 3?`

Additional Problems

DIRECTIONS: Answer each problem below to the best of your ability. Solutions are not provided in this book.

Problem 28:

Write a program that performs the following tasks:

1. The user enters a command line argument for an integer `n`.
2. If the number is negative or 0, the program will terminate.
3. The program will calculate the number of 7s with the use of a method called **luckySeven** that takes an integer parameter.

Problem 29:

Write a program that performs the following tasks:

1. Prompt the user to enter at least a positive integer of at least even length (2 digits, 4 digits, etc.).

2. If the user does not enter a correct value, the program will terminate.
3. With the use of a method called **diffNeighbors**, return true if every other digit its integer parameter are greater than or equal to its previous. Example is:

`diffNeighbors(123456) = returns true`

`diffNeighbors(680046) = returns true`

`diffNeighbors(8647) = returns false`

Problem 30:

Write a program that performs the following tasks:

1. Prompt the user to enter a positive 5 digit integer.
2. If the user does not enter a correct value, the program will terminate.
3. Write and utilize a method called **trip8**, which takes the entered number as its parameter, that will print the number, triplicating any digit that is an 8. An example is:

`trip8(67838) = prints 678883888`

`trip8(12345) = prints 12345`

Problem 31:

Write a program that performs the following tasks:

1. Prompt a user to enter a positive 6 digit integer (the variable for entry will be called *num*).
2. If the entered value is not correct, the user enters the value again until it is correct.
3. Prompt the user to enter a positive integer between 1 and 5 (the variable for entry will be called *X*).
4. If the entered value is not correct, the program will assign a value of 2 to *X*.
5. With the use of a method called **downX**, taking two parameters reflecting *num* and *X*, print a returned value from the method that reflects the decrease of each digit of *num* by *X*. If the digit decrease will make a value of 0 or negative, do not add that to the final result. An example is below:

`downX(523412, 3) = returns 21`

`downX(544436, 5) = returns 1`

Problem 32:

Write a program that performs the following tasks:

1. The user enters 2, equal length integers from the command line.

2. If the digits are not of equal length, the program terminates.
3. With the use of a method called **digitDiff**, that takes two parameters, print the difference of each digit to the console. An example is:

`digitDiff(5544, 1244)` prints 4300

(5-1, 5-2, 4-4, 4-4)

`digitDiff(12345, 43215)` prints -3-1130

(1-4, 2-3, 3-2, 4-1, 5-5)

Problem 33:

Write a program that performs the following tasks:

1. The user enters 2, equal length integers from the command line.
2. If the digits are not of equal length, the program terminates.
3. With the use of a method called **digitProds**, that takes two parameters, print a returned value, representing the product of each digit, to the console. An example is:

`digitProds(554, 144)` returns 52016

(5*1, 5*4, 4*4)

`digitProds(12345, 43215)` returns 466425

(1*4, 2*3, 3*2, 4*1, 5*5)

Problem 34:

Write a program that performs the following tasks:

1. The user enters a positive 6 digit integer.
2. If the entry is incorrect, the program terminates.
3. The user enters an integer between 0 and 9.
4. If that entry is incorrect, the program terminates.
5. With the use of a method called **lastOccurrence**, which takes two integer parameters (the first is the 6 digit integer and the other is the single digit integer), return a value representing the position of the last occurrence of the second parameter in the first parameter. If the second parameter is not contained in the first parameter, return a value of -1.
6. With the use of a method called **firstOccurrence**, taking the same two parameters as above, return a value representing the position of the first occurrence of the second parameter in the first parameter. If the second parameter is not contained in the first parameter, return a value of 99.

`lastOccurrence(654565, 5)` returns a value of 6

`firstOccurrence(654565, 5)` returns a value of 2

`lastOccurrence(555444, 2)` returns a value of -1

`firstOccurrence(555444, 2)` returns a value of 99

Problem 35:

Write a program that performs the following tasks:

1. The user enters 4 positive integer values, a, b, c, x.
2. If the x entry is not positive, the program terminates.
3. With the use of a method called **quadratic**, return a value representing a quadratic with the following formula: $ax^2 + bx + c$

Problem 36:

Write a program to perform the following tasks:

1. The user enters a 7 digit positive integer from the command line.
2. If the integer is not correct or is negative, the program terminates.
3. With the use of a method called **barGraph**, taking an integer argument, print to each row a value reflecting the numbers from 1 to the digit of the integer parameter. If the digit is 0, nothing should print on a row. An example is:

```
barGraph(4217024)
```

```
1234
```

```
12
```

```
1
```

```
1234567
```

```
12
```

```
1234
```

Problem 37:

Write a program to perform the following tasks:

1. The user enters a 10 digit positive integer from the command line.
2. If the integer is not correct or is negative, the program terminates.
3. With the use of a method called **barGraphChar**, taking an integer argument, print a 10 by 10 grid of the * character, where each row prints a certain number of * characters reflecting the value of the digit of the integer parameter. If the digit is 0, nothing should print on a row. White space characters should print in the beginning of the row, reflecting a value of 10 – digit. An example is:

```
barGraphChar(4217024057)
```

```
*****
 **
 *
 *****
 *
 *
```

Problem 38:

Write a program that performs the following tasks:

1. Asks a user for a positive integer x .
2. If the number entered is less than 0 or larger than 10, the program terminates.
3. With the use of a method called **equalSum**, print out the numbers in the range from 1 to x^2 who's sum of the digits are equal to x .

Sample output:

Enter a positive integer x: 4

4

13

Problem 39:

Write a program that performs the following tasks:

1. Asks a user for a positive integers x and y .
2. If either number entered is less than 0 or larger than 25, the program terminates.
3. With the use of a method called **prodSum**, print out the numbers in the range from 1 to x who's product of the digits are equal to y .

Sample output:

Enter 2 integers x and y: 25 4

4

14

22

Problem 40:

Write a program that performs the following tasks:

1. Asks a user for a positive integers a and b , where b is in the range of 0 to 9.
2. If the entry for b is incorrect, prompt the user to reenter it until correct.
3. With the use of a method called **isLarger**, print out the numbers in the range from 1 to a where each digit in the number is at least as large as y .

Sample output:

100 8

8

9
88
89
98
99

Problem 41:

Write a program that performs the following tasks:

1. Ask a user for two, equal length positive integers.
2. If the integers are negative or not of equal length, the program terminates.
3. With the use of a method called **gapSum**, return a value representing the sum of the gaps of each digit.

Sample output:

Enter 2 positive integers of equal length: 61 16

10

Problem 42:

Write a program that performs the following tasks:

1. Ask a user for two, equal length positive integers.
2. If the integers are negative or not of equal length, the program prompts the user for reentry until correct.
3. With the use of a method called **gapProducts**, return a value representing the product of the gaps of each digit.

Sample output:

Enter 2 positive integers of equal length: 61 16

25

Problem 43:

Write a program to perform the following tasks:

1. Ask a user to enter 3, positive 5 digit integers.
2. If any of the 3 integers are out of range, the program terminates.
3. With the use of a method named **beginsWith**, the program prints whether the starting value is even or odd.

Sample output:

123

456

odd
even
odd

Problem 44:

Write a program that performs the following tasks:

1. Ask the user to enter a String value of at least length 5.
2. If the entry is not of at least length 5, the program asks for another word until correct.
3. The program then prints out a large version of the most frequent vowel found in the word. If the word does not contain at least 1 vowel, nothing will print.

Sample output:

Enter a word: Eagle

```
*****  
*  
***  
*  
*****
```

CHAPTER 9

Recursion

This chapter covers the basics of recursive methods (methods that call itself until a condition is met). Some classic examples of recursion (Factorials and Fibonacci numbers) are shown.

TOPICS

1. <i>Recursive Tracing</i>	176
2. <i>The Factorial Method</i>	180
3. <i>The Fibonacci Method</i>	182
4. <i>Exercises</i>	183

By definition, a **recursive** method is a method that calls itself until a certain condition is met. That condition is called the **base case**.

The best practice of recursion is to dive right in, see a lot of examples, and practice.

RECURSIVE TRACING

When recursive methods are being called, they are placed on what is called the **runtime stack** of a program. Just the same as a stack of dishes, the item on the top is the item that is accessed. With a program, it is the same idea that the most recent call to the stack (or the top of the stack) is the currently active method call. Let's see a small example.

EXAMPLE 1: Recursive Printing

Here is an example that will recursively print out the numbers from 1 to 5.

```
public class Example1{
    public static void Print(int n){

        //base case
        if(n < 1) return;

        Print(n-1); //recursive call

        System.out.println(n);

    } //Print

    public static void main(String args[]){
        Print(5);
    } //main
} //class
```

The Print() recursive method in the above program will accept an integer argument. The base case of the method will be when the argument is 0, since we want a number from 1 to 5. It will simply return to the last method call.

Then, if the base case is false, the method is called again with 1 less for the argument value. If you did not do this, **the recursion would be infinite**.

Finally, it will simply output that current value of the argument when the recursive calls are finished.

Here is the trace for the above printing example. It shows the call of the method and the value of n.

```
Print(5)
Print(4)
Print(3)
Print(2)
Print(1)
    Print(0) ← base case here.
    Print(1) ← prints 1
    Print(2) ← prints 2
    Print(3) ← prints 3
    Print(4) ← prints 4
Print(5) ← prints 5
```

Another example is to print out a triangle of stars recursively:

EXAMPLE 2: *Recursive Triangle of Stars I*

```
public class Example2{
    public static void Triangle(int x) {
        //base case
        if (x <= 0) return;

        //recursive call
        Triangle(x - 1);

        //print the stars with a loop
        for (int i = 1; i <= x; i++)
            System.out.print("*");

        System.out.println();
    } //Triangle method

    public static void main(String args[]) {
        Triangle(7);
    } //main
} //class
```

The trace of the program will be as follows:

```
Triangle(7)
Triangle(6)
Triangle(5)
Triangle(4)
Triangle(3)
Triangle(2)
Triangle(1)
    Triangle(0) ← base case
    Triangle(1) ← prints 1 star & new line
    Triangle(2) ← prints 2 stars & new line
    Triangle(3) ← prints 3 stars & new line
    Triangle(4) ← prints 4 stars & new line
    Triangle(5) ← prints 5 stars & new line
    Triangle(6) ← prints 6 stars & new line
Triangle(7) ← prints 7 stars and new line
```

The output will be:

```
*
```

```
**
```

```
***
```

```
****
```

```
*****
```

```
******
```

EXAMPLE 3: Recursive Triangle of Stars II

This version of the triangle of stars shows the reverse of the above, where the method will print a row of stars first, and then make a recursive call.

```
public class Example3{
    public static void Triangle(int x) {
        //base case
        if (x <= 0)
            return;

        //print the stars with a loop
        for (int i = 1; i <= x; i++)
            System.out.print("*");

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

```

        //recursive call
        Triangle(x - 1);
    } //Triangle method
    public static void main(String args[]) {
        Triangle(7);
    } //main
} //class

```

The trace of the program will be as follows:

```

Triangle(7) ← prints 7 stars and new line
Triangle(6) ← prints 6 stars & new line
Triangle(5) ← prints 5 stars & new line
Triangle(4) ← prints 4 stars & new line
Triangle(3) ← prints 3 stars & new line
Triangle(2) ← prints 2 stars & new line
Triangle(1) ← prints 1 star & new line
Triangle(0) ← base case
Triangle(1)
Triangle(2)
Triangle(3)
Triangle(4)
Triangle(5)
Triangle(6)
Triangle(7)

```

The output will be:

```

*****
 *****
 ****
 ***
 **
 *

```

EXAMPLE 4: Sum from 1 to N

This version of finding the sum from 1 to N will make use of a recursive method. The user enters the value for N.

```

import java.util.Scanner;
public class Example4 {
    //recursive method for finding the sum
    private static int sum(int n) {
        if(n <= 1) return 1; //base case
        return n + sum(n-1); //otherwise call again
    }
    public static void main(String args[]){
        Scanner s = new Scanner(System.in);
        int n = 0;

        //prompt and get entry
        System.out.println("Enter a number > 1: ");
        n = s.nextInt();

        //call method
        System.out.println( sum(n) );
    } //main
} //class

```

Let's say the user enters the value of 5. The trace of the program would be as follows:

```

sum(5)
5 + sum(4)
  4 + sum(3)
    3 + sum(2)
      2 + sum(1)
        1      ← base case
      2 + 1
    3 + 3
  4 + 6
5 + 10
15

```

The two most classic types of recursive methods are the **Factorial** method and the **Fibonacci** method.

THE FACTORIAL METHOD

In math, the factorial of an integer is defined as the product of the numbers less than the given integer up to 1. So, say we wish to find the factorial of 4 (written as 4!) it is written out as follows:

$$4! = 4 \times 3 \times 2 \times 1 = 24$$

In Java, we can write a recursive method to do the same thing. Below is a short program containing that method, and a proof that it works:

EXAMPLE 5: *Recursive Factorial*

```
public class Example5{
    public static int fact( int n ){
        //base case:
        if( n <= 1 ) return 1;

        //recursive call
        return n * fact(n-1);
    }
    public static void main(String args[]){
        System.out.println(fact(4));
    } //main
} //class
```

Proof Recursively:

In the above main method, we are trying to find the factorial of the integer 4. Using the rule of the factorials, the answer should be 24, since $4 \times 3 \times 2 \times 1 = 24$. Applying the method and passing in 4 as the parameter, here is the tree:

```
Fact(4)
  4* Fact(3)
    3* Fact(2)
      2* Fact(1)
        2* 1 → 2
      3* 2 → 6
    4* 6 → 24
```

The recursive method runs and works properly.

The Iterative Factorial method

(No recursion). This is obviously one of many ways to solve this problem.

```
public static long fact( long n ){
    long ans = 1;
```

```

        for(long i = n; i > 1; i--)
            ans *= i;

    return ans;
}

```

THE FIBONACCI METHOD

The famous Fibonacci sequence is generated by finding the sum of its previous two numbers. For example, the first two Fibonacci numbers are 1 and 1. Therefore, the next Fibonacci number is 2, since $1 + 1 = 2$. Below are the first few numbers in the sequence:

1 1 2 3 5 8 13 21 ...

Below is a program that will calculate the n^{th} Fibonacci number, and a proof that the method will work:

EXAMPLE 6: *Recursive Fibonacci*

```

public class Example6{
    public static int fib( int n ){
        //base case:
        if( n <= 2 ) return 1;

        return fib(n-2) + fib(n-1);
    } //fib

    public static void main(String args[]){
        System.out.println(fib(3));
    } //main
} //class

```

Proof Recursively:

Assume you wish to know the 3rd Fibonacci number in the famous sequence. The third number of the sequence is 2 ($1 + 1 = 2$). Now, apply the Fibonacci recursive method still wishing to find the 3rd number. Below is the tree step by step:

```

        fib(3)
        /   \
fib(3-2) + fib(3-1)

```

```
    fib(3)
    /   \
fib(1) + fib(2)
```

```
    fib(3)
    /   \
1     +   1
```

The base cases applied to the fib(1) and the fib(2).

The Iterative Fibonacci method:

(No recursion). This is one of many ways to solve this problem.

```
public static long fib( long n ){
    long n1 = 1, n2 = 1;

    for(int i = 3; i <= n; i++){
        long c = n1 + n2; //sum of previous 2
        n1 = n2; //old n2
        n2 = c; //new number is the sum
    }

    return n2;
}
```

EXERCISES

DIRECTIONS: Answer each question below to the best of your ability. Solutions on page 616.

Writing Recursive Methods

Problem 1:

Using your knowledge of recursion, write a method called boxes, taking as parameters, an integer x and a character y. The method will output an x by x box of characters. So, an example run of the program would be:

```
*****
*****
*****
*****
*****
```

For $x = 5$ and $y = '*'$

You can assume a variable t within the program to hold the number of the recursive call.

Problem 2:

Write a recursive method that will output a triangle of 0s and 1s in rotating fashion from large to small. The method, called triangles(), will take an integer argument, as that is the number of rows in the triangle.

Sample output:

```
11111  
0000  
111  
00  
1
```

Problem 3:

Rewrite problem 2 to print out the triangle from smallest to largest. The method is still recursive.

Sample output:

```
1  
00  
111  
0000  
11111
```

Problem 4:

Write a recursive method that will print out the even numbered characters of a string in reverse.

Sample input:

“hello”

Sample output:

le

Problem 5:

Write a recursive method to find the largest digit of an integer passed to it.

Sample input:

12845673

Sample output:

8

You can assume a variable called lg as part of the program to hold the current largest value.

Problem 6:

Write a recursive method that will reverse the digits of an integer that is passed to it.

Sample input:

1284

Sample output:

4821

Problem 7:

Write a recursive method that will print out the reverse of the digits of an integer passed to it.

Sample input:

1284

Sample output:

15

Problem 8:

Write a recursive method that will count the number of digits of an integer passed to it.

Sample input:

451342

Sample output:

6

Converting Recursive Methods

Problem 9 to 16:

Convert each recursive method from the above problems 1 through 8 into an iterative solution (non-recursive solution). Keep the method name the same, as well as any arguments needed.

Recursion Output Tracing

Problem 17:

```
public class Problem17{
    public static void main(String args[]){
        printSome(????);
    } //main

    public static void printSome(int i){
        int k = i;

        if(k <= 0 || i <= 0) return;

        for(int j = k-1; j <= 3; j+=2)
            System.out.print(j);

        printSome(k-1);

    } //printSome
} //class
```

1. What is the output given the following inputs to the method:

- a. 3
- b. 5
- c. 8

Problem 18: (variation on Problem 17)

```
public class Problem18{
    public static void main(String args[]){
        printSome2(????);
    } //main

    public static void printSome2(int i){
        int k = i;

        if(k <= 0 || i <= 0) return;

        for(int j = k-1; j <= 3; j+=2)
            System.out.print( (j+k) );

        printSome2(k-1);

    } //printSome
} //class
```

1. What is the output given the following inputs to the method:

- a. 3

- b. 5
- c. 8

Problem 19: (variation on Problems 17 & 18)

```
public class Problem19{  
    public static void main(String args[]){  
        System.out.println( printSome3(??? ) );  
    } //main  
  
    public static int printSome3(int i){  
        int k = i;  
  
        if(k <= 0 || i <= 0) return;  
  
        for(int j = k-1; j <= 3; j+=2, --i)  
            System.out.print( (j+k) );  
  
        return i+ printSome3(k-1);  
    } //printSome3  
} //class
```

1. What is the output given the following inputs to the method:

- a. 3
- b. 5
- c. 8

Problem 20:

```
public class Problem20{  
    public static int magic(int n){  
        if(n == 10) return n--;  
  
        System.out.print(2*n + "\n");  
  
        return magic(++n);  
    } //magic  
  
    public static void main(String args[]){  
  
        int n = 1;  
        magic(n);  
  
    } //main  
} //class
```

1. What will be the output of the program?

2. If the last line of the magic() method was changed to:

```
return magic(n++)
```

Explain what the output would be.

Problem 21:

```
public class Problem21{
    static long x(long y){
        if(y < 10)
            return y+10;

        System.out.print( y + " " );
        x(y-1);
        System.out.print( y + " " );
        return 1;
    }
    public static void main(String args[]){
        for(int i = 5; i < 25; i += 4){
            System.out.println( x(i) );
        }
    } //main
} //class
```

1. What is the output of the above program?

Problem 22:

```
public class Problem22{
    static int z(int x, int y){
        if(x < y)
            return 1;
        z(x-2, y-1);
        System.out.print(x-y + " ");
        return y;
    }
    public static void main(String args[]){
        //for z method
        int c = z(9, 7);
        int d = z( z(4, 9), 6 );
        int e = z( z(9,9), z(12, 10) );

        System.out.println(c + " " + d + " " + e);
    } //main
} //class
```

1. What is the output of the above program?

Problem 23:

```
public class Problem23{
```

```

public static void f(long x, long i){
    if(x % i == 0)
        return;

    f(x-2, i+1);
    System.out.println(x + " " + i);
}

public static void main (String[] args){
    long a = 8, b = 6;

    f(a, b);
} //main
}

```

1. What is the output of the above program?

Problem 24:

```

public class Problem24{
    public static void f(int x){
        System.out.println(x);

        if(x == 0) return;

        f(x-1);

        System.out.println(x);
    }

    public static void main (String[] args){
        f(6);
    } //main
}

```

1. What is the output of the above program?

Problem 25:

```

public class Problem25{
    public static void p(int a, int b){
        if(a == 0) return;

        p(a-1, b);

        for(int i = 0; i < a; i++)
            System.out.print(b);
        System.out.println();
    }

    public static void main (String[] args){

```

```
    p(4, 7);
} //main
}
```

1. What is the output of the above program?

Problem 26: (variation on Problem 25)

```
public class Problem26{
    public static void p(int a, int b){
        if(a == 0) return;

        for(int i = 0; i < a; i++)
            System.out.print(b);
        System.out.println();

        p(a-1, b);
    }

    public static void main (String[] args){
        p(4, 7);
    } //main
}
```

1. What is the output of the above program?

Problem 27:

```
public class Problem27{
    public static void f(int n){
        if(n == 0) return;

        f(n-1);

        System.out.println(n*n);
    }

    public static void main (String[] args){
        f(7);
    } //main
}
```

1. What is the output of the above program?

Problem 28:

```
public class Problem28{
    public static void x(int y, int z){
        if(y + z == 10) return;

        System.out.println((y + z));
    }
}
```

```

        x(y-1, z);
    }

    public static void main (String[] args){
        int n = 5;
        x(n, n+5);
    } //main
}

```

1. What is the output of the above program?

Problem 29:

```

public class Problem29{
    public static int t(int a, int b){
        if((a+b) < 5)
            return 0;

        System.out.print( t(a-1, b-2) );
        return a+b;
    }

    public static void main (String[] args){
        int n = 5;
        System.out.println( t(n, 10) );
    } //main
}

```

1. What is the output of the above program?

Problem 30:

```

public class Problem30{
    static void f(int n, char c){
        if(n <= 0) return;

        for(int i = 0; i < n; i++)
            System.out.print(c);

        System.out.println();
        f(n-1, c);
    }

    public static void main (String[] args){
        f(5, '+');
    } //main
}

```

1. What is the output of the above program?

Problem 31:

```
public class Problem31{
    static int p(int n, int m){
        if(n < m) return n;

        System.out.println(m);

        p(n-2, m-1);

        System.out.println(n);

        return n;
    }
    public static void main (String[] args){
        int x = p(15, 10);
        System.out.println(x);
    }//main
}
```

1. What is the output of the above program?

Problem 32:

```
public class Problem32{
    static void p(int x){
        if(x <= 0) return;

        p(x-1);

        System.out.print(x);

        p(x-1);
    }
    public static void main (String[] args){
        p(3);
    }//main
}
```

1. What is the output of the above program?

Problem 33:

```
public class Problem33{
    static void p(int x){
        if(x <= 0) return;

        p(x-1);
        p(x-1);
    }
}
```

```

        System.out.print(x);
    }
    public static void main (String[] args){
        p(3);
    } //main
}

```

1. What is the output of the above program?

Problem 34:

```

public class Problem34{
    public static int Q(int a, int b){
        if(a < b) return 0;
        else if(a > b) return a;

        return 3 + Q(a-1, b-2);
    }
    public static void main (String [] args){
        int a = 0, b = 4;
        System.out.println(Q(a, b));
        System.out.println(Q(b, a));
        System.out.println(Q(Q(b, a), b));
    } //main
} //class

```

1. What is the output of the above program?

Problem 35:

```

public class Problem35{
    public static int F(int c){
        if(c < 0) return c*-1;
        else if(c >= 0) return c+1;

        return c + c*-1 + F(c-2);
    }
    public static void main (String [] args){
        System.out.println(F(0));
        System.out.println(F(9));
        System.out.println(F(F(3)));
        System.out.println(F(F(F(3))));
    } //main
} //class

```

1. What is the output of the above program?

Problem 36:

```

import java.util.Scanner;
public class Problem36{
    public static int H(int y){
        if(y == 0) return 1;
        else return 3 + H(y-1);
    }
    public static int HH(int x){
        int z = H(x);
        z = z + z - 1;
        return z;
    }
    public static void main (String [] args){
        Scanner s = new Scanner(System.in);
        int x = s.nextInt();

        int z = 0;
        if(x % 2 == 0)
            z = HH(x);
        else
            z = H(x);

        System.out.println("z= " + z);
    } //main
} //class

```

1. What is the output of the above program when the user enters the following values for x:

- a. x = 2
- b. x = 3
- c. x = 10

Additional Problems

DIRECTIONS: Answer each problem below to the best of your ability. Solutions are not provided in this book.

Problem 37:

Write a program that performs the following tasks:

1. Ask the user for an integer n.
2. If the number is negative or 0, the program will exit.
3. The program will calculate the product of digits with the use of a recursive method **prodDigits()** that takes an integer parameter.

Problem 38:

Write a program that performs the following tasks:

1. The user enters a command line argument for an integer n .
2. If the number is negative or 0, the program will exit.
3. The program will calculate the sum of the odd digits with the use of a recursive method **sumOdd()** that takes an integer parameter.

Problem 39:

Write a program that performs the following tasks:

1. The user enters a command line argument for an integer n .
2. If the number is negative or 0, the program will exit.
3. The program will calculate the number of 7s with the use of a recursive method **luckySeven()** that takes an integer parameter.

Problem 40:

Write a program that performs the following tasks:

1. The user enters a command line argument for an integer n followed by a String character c .
2. If the number is negative or 0, the program will exit.
3. The program will print the character c , n times on n lines with the use of a recursive method **printChar()** which takes an integer argument and a String argument.

Problem 41:

Write a program that performs the following tasks:

1. The user enters a command line argument for a String called *word*.
2. The program will calculate the number of vowels that are uppercase with the use of a recursive method **vowelCount()** that takes a String parameter.

Problem 42:

Write a program that performs the following tasks:

1. The user enters a command line argument for an integer n .
2. If the number is negative, the program will exit.
3. The program will calculate and print the binary equivalent of the number with a recursive method **toBinary()**. The method takes an integer parameter.

Problem 43 to 48:

Convert each recursive method from the above problems 37 through 42 into an iterative solution (non-recursive solution). Keep the method name the same, as well as any arguments needed. The structure of the programs stay the same.

CHAPTER 10

Arrays

This chapter covers one of the most important and widely used topics in Java. It shows both a one-dimensional and two-dimensional array in addition to lots of examples that will greatly enhance your understanding.

TOPICS

1. <i>One-Dimensional Arrays</i>	197
2. <i>Length Method</i>	199
3. <i>One-Dimensional Array Examples</i>	200
4. <i>One-Dimensional Array Exercises</i>	204
5. <i>Two-Dimensional Arrays</i>	213
6. <i>Two-Dimensional Array Exercises</i>	217
7. <i>Foreach Loops</i>	222
8. <i>Arrays & Methods</i>	226
9. <i>Arrays & Methods Exercises</i>	233

By definition, an **array** is collection of data that is side by side in memory. Like any variable, an array must be declared before it is used. There are also numerous types of arrays.

In Java, an array is treated **as an object**. It can require the keyword **new** to create an array of a certain type. An array is also an ordered list of values.

ONE-DIMENSIONAL ARRAYS

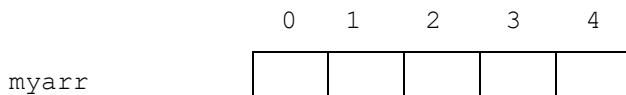
In general, this is how to declare a one-dimensional array:

```
type name[] = new type[ size ];
```

Where in the above, **type** is the appropriate data type; **name** is the name of the array; and **size** is how many pieces of data you want inside it.

```
int myarr[] = new int[5];
```

The above would declare an array of 5 integers. In memory, here is what it would look like:



Where the numbers above the cells are the **index**. The index is where that particular piece of data is located. All array indices begin at **zero** and end at the **size-1**. So above, the capacity is 5, but the indices range from 0 to 4.

Say we wanted to fill the array with numbers. Here is a code snippet using a for loop:

```
for(int i = 0; i < 5; i++)  
    myarr[i] = i * 2;
```

This loop will place doubled value of the loop counter inside the array.

The array is accessed with the **[] operator**. What goes inside of the brackets is the index of the array. The code below is incorrect and a common mistake made by programmers:

```
for(int i = 1; i <= 5; i++)
    myarr[i] = i * 2;
```

The reason it is wrong is that there is no index 5 in the array we declared. You would get something called an **exception**, specifically `ArrayIndexOutOfBoundsException` (see Chapter 15), which is the Java way of saying you are out of bounds in the array. The above is also ignoring index 0.

Here are just some more declarations so you can see how to do it in Java:

```
int heights[] = new int[5];

float f[] = new float[15];

char alphabet[] = new char[26];

boolean bools[] = new boolean[45];

String str[] = new String[10];
```

You can also declare an array and initialize it at the same time with the use of what's called an **initializer list**. This will not involve the `new` operator and can be seen like the below:

```
int heights[] = {1, 2, 3, 4, 5};
This would declare and initialize an integer array of size 5,
containing indices 0 - 4.

char alphabet[] = {'A', 'B', 'C', 'D', 'E'};
This would declare and initialize a character array of size 5,
containing indices 0 - 4.

boolean bools[] = {true, false, false, false, true, true, true};
This would declare and initialize a boolean array of size 7, containing
indices 0 - 6.

String str[] = {"One", "Two", "Three", "Four"};
```

This would declare and initialize a String array of size 4, containing indices 0 - 3.

There are some shorthand ways of declaring and initializing arrays. Let's see how to do some declarations. These two declarations are the same:

```
int arr[];  
int[] arr;
```

Now, let's say you have more than one array you need to declare. These two below are the same:

```
int[] a, b, c;  
int a[], b[], c[];
```

LENGTH METHOD

A method that is built into arrays is called the **length()** method. This will return an integer representing the physical size of the array (that DOES NOT MEAN SIZE-1). Let's say we have this declaration:

```
int[] a = {3,4,5,9,10};
```

And we call the length method when dealing with a for loop:

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

The value that will be returned from the length() method is 5, since the actual size of the array is 5 (indices 0-4).

ONE-DIMENSIONAL ARRAY EXAMPLE PROGRAMS

Below are short programs demonstrating one-dimensional arrays.

EXAMPLE 1: *One-dimensional Basics*

This program will declare an int array of size 10, and fill it with values through the use of a for loop. It then outputs what the array contains.

```
public class Example1{
    public static void main(String args[]){
        //1d array of integers of size 10 (indexes 0 to 9)
        int myarr[] = new int[10];

        //give the array values by looping from 0 to 9
        for(int i = 0; i < 10; i++)
            myarr[i] = i % 3;

        //display results
        for(int j = 0; j < 10; j++)
            System.out.print(myarr[j] + " ");
    }//main
} //class
```

The output when running the above program is:

0 1 2 0 1 2 0 1 2 0

EXAMPLE 2: *One-dimensional Sum*

This program will declare an array of size 5 of type double, and fill it with values entered by the user. It then outputs the sum of the values in the array.

```
import java.util.Scanner;
public class Example2{
    public static void main(String args[]){
        //String variable
        Scanner s = new Scanner(System.in);

        //variables needed
        double entries[] = new double[5];
```

```

        double sum = 0.0;
        int count = 0;

        //get the 5 numbers from the user
        while (count < 5) {
            System.out.println("Please enter a decimal: ");
            entries[count] = s.nextDouble();
            count++;
        }

        //find the sum of the numbers
        for(int i = 0; i < 5; i++)
            sum += entries[i];

        System.out.println("The sum is: " + sum);
    } //main
} //class

```

A sample run of the above program is:

```

Please enter a decimal: 5
Please enter a decimal: 4
Please enter a decimal: 3.3
Please enter a decimal: -3.3
Please enter a decimal: 0
The sum is: 9.0

```

EXAMPLE 3: *One-Dimensional Even Numbers*

Below is a program that will count the number of even numbers in a given array of integers. The user will enter the values, which are then stored in the array. The output is the count of even and odd numbers entered in the array.

```

import java.util.Scanner;
public class Example3{
    public static void main(String args[]){
        System.out.println("Please enter 10 positive numbers: ");

        //declare array of size 10 and declare Scanner
        int arr[] = new int[10];
        Scanner s = new Scanner(System.in);

        //get the number from the user in the console
        for(int i = 0; i < 10; i++){

```

```

        arr[i] = s.nextInt();
    }

    //loop to count the even and odd numbers entered
    int evens = 0, odds = 0;

    for(int i = 0; i < 10; i++){
        if(arr[i] % 2 == 0) evens++;
        else odds++;
    }

    //display result
    System.out.println("There are " + evens
        + " even numbers and " + odds
        + " odd numbers.");
} //main
} //class

```

A sample run of the above program may be:

Please enter 10 numbers, all positive:

5
6
7
8
9
10
4
2
3
2

There are 6 even numbers and 4 odd numbers.

EXAMPLE 4: Vertical Bar Graph with Arrays

This program will simply ask the user for 10 numbers, and print 10 rows of stars as specified by the user. The array stores the numbers from the user and represents the size of a particular line.

```

import java.util.Scanner;
public class Example4{
    public static void main(String args[]) {

```

```

//declare array of size 10
int arr[10] = new int[10];

//declare Scanner for input
Scanner s = new Scanner(System.in);

System.out.println("Please enter 10 numbers:");
//get the 10 values from the user
for(int i = 0; i < 10; i++)
    arr[i] = s.nextInt();

int i = 0;
while(i < 10){
    //print the specified number of stars
    for(int j = 0; j < arr[i]; j++)
        System.out.print("*");

    System.out.println();
    i++;
}
} //main
} //class

```

A sample run of the above program may be:

Please enter 10 numbers:

5 4 3 2 1 0 1 2 3 4

**

*

*

**

EXAMPLE 5: *String Arrays*

Here is small program to show both a declaration and creation of Strings:

```

public class Example5{
    public static void main(String[] args){
        String str[] = new String[5];

        for(int i = 0; i < str.length; i++)
            str[i] = new String("Hello " + i);

        //perform a small change:
        str[0] = str[3];

        for(int i = 0; i < str.length; i++)
            System.out.println(str[i]);
    } //main
} //class

```

The first for loop will actually create the String object (hence the keyword new and the constructor call). Each string will be "Hello i," where i is the loop counter. After a small change in array (done by assigning the value of index 3 to index 0, the array is printed out. The output from running the above program is:

Hello 3
 Hello 1
 Hello 2
 Hello 3
 Hello 4

ONE-DIMENSIONAL ARRAY EXERCISES

DIRECTIONS: Answer each question below to the best of your ability. Solutions on page 624.

Problem 1:

Declare a 1 dimensional array of the following:

- An array of short integers called *shorts* of size 20.
- An array of integers called *nums* of size 100.
- An array of floats called *grades* of size 11.
- An array of double precision numbers called *numsDbl* of size 50.
- An array of char called *characters* of size 100.
- An array of type String called *student_names* of size 22.

Problem 2:

Initialize each of the above arrays using any technique you wish (solutions will vary from ours).

Problem 3:

Write a *for loop* that will perform each of the following tasks.

- a) Initialize a 1 dimensional char array called **arr** to all 'B' of size 15.
- b) Print each element of a 1 dimensional integer array called **iarr** of size 20.
- c) Print each element of a 1 dimensional float array called **far** with 20 elements.
- d) Decrement each element of a 1 dimensional long array called **larr** of size 100.
- e) Print the total number of occurrences of the character 'B' or 'b' in a char array called **car** of size 25.

Problem 4:

Write a *while loop* to perform each of the following tasks.

- a) Initialize a 1 dimensional char array called **arr** to all 'X' of size 25.
- b) Print each element of a 1 dimensional long array called **iarr** of size 125.
- c) Print each element of a 1 dimensional float array called **far** with 20 elements.
- d) Decrement each element of a 1 dimensional long array called **larr** of size 100.
- e) Print the total number of occurrences of the character 'B' or 'b' in a char array called **car** of size 25.

Writing Programs

Problem 5:

Write a program to perform the following tasks relating to a 1 dimensional array of integers:

1. Find and display the lowest number in the array
2. Find and display the highest number in the array
3. Find and display the sum of all the numbers in the array
4. Find and display the average of all the numbers in the array

The array will look like this:

```
int arr[] = {1, 2, 6, 3, 45, 6, 5, 3, 7, 9, 7, 65, 7, 9, 0};
```

Sample output:

The min value is: 0

The max value is: 65

The total sum is: 175

The average is: 11.66666666666666

Problem 6:

Write a program to perform the following tasks:

1. Count how many times the first element in an array appears in an entire array.
2. Count how many times the second element appears in the whole array.
3. If the result of the first task is larger than the result of the second task, all values of the array are doubled. If the second is larger, the values are decreased by 5 in the entire array. If they are equal, nothing will happen.
4. Display the array by printing each element.

The array will look like this:

```
short arr[] = {2,5,2,2,5,3,2,3,5,3,4,5,8,5,5,2};
```

Sample output:

```
-3 0 -3 0 -2 -3 -2 0 -2 -1 0 3 0 0 -3
```

Problem 7:

Write a program to perform the following tasks:

1. Count the number of ones in any position of a number in a 1 dimensional array.
2. Display the total count for the entire array.

The array will look like this:

```
short nums[]={18, 912, 121, 300, 601, 2121, 3122, 1991, 12, 1001};
```

Sample output:

There are 13 ones in the array.

Problem 8:

Write a program to perform the following tasks:

1. Prompt the user to enter a positive integer x greater than or equal to 0.
2. If the entry is not positive, the program terminates.
3. Given the String array below, print the x numbered character in each element of the array. If the entry is longer than the length of the String, a white space will print.

```
String animals[] = {"Dog", "Cat", "Lion", "Tiger", "Lizard", "Bear",  
"Penguin", "Gorilla"};
```

Sample output:

Enter an integer >= 0: 0

DCLTLBPG

Problem 9:

Write a program to perform the following tasks:

1. Prompt the user to enter a positive integer x between 10 and 25 inclusive.
2. If the entry is not correct, the user will be prompted to try again until they get the entry correct.
3. Given the integer array below, print the values of the array, plus or minus the value entered, from the first element in the array. The first element in the array will not be printed.

```
int arr[] = {100,221,68,90,95,96,85,73,77,79,77,65,97,99,101};
```

Sample output:

Enter a positive integer x: 25

90 95 96 85 77 79 77 97 99 101

Output Tracing

Problem 10:

For each of the following code snippets below, what is the output?

a)

```
int arr[] = {3,4,5,6,7};
for(int i = arr[0]; i < arr[2]; i++){
    System.out.print((arr[i] + arr[0]));
}
```

b)

```
float f[] = {9.0,8.0,7.0,6.0};
for(int i = 0; i < 4; i++){
    System.out.print(f[i]*f[i] + " " );
}
```

c)

```
char ch[] = {67,68,69,'A','E'};
for(int i = 0; i < 5; i++)
    System.out.print(ch[i] + " " );
```

d)

```
char str[] = {'A','B','C','D'};
for(int i = 0; i < 8; i++) {
```

```

        int p1 = i%4, p2 = (i+1) % 4;
        System.out.print(str[p1] + str[p2] + " ");
    }

e)
String str[] = {"Hello", "World", "I", "Am", "Great"};
for(int i = 0; i < 5; i++)
    System.out.print( str[i].charAt(i % 2) );

f)
String str[] = {"Hello", "World", "I", "Am", "Great"};
for(int i = 0; i < 10; i++)
    System.out.print( str[i%2].charAt(i % 2) );

```

Problem 11:

What is the output of the below program?

```

public class Problem11{
    public static void main (String[] args){
        int x[] = {7, 4, 0, 2, 3, 1, 6};

        for(int i = 0; i < 7; i++){
            System.out.print(x[i]);

            switch(x[i]){
                case 0:
                case 1:
                case 2:
                case 3:
                case 4:
                case 5:
                case 6:
                    System.out.println("N");
                    break;
                case 7:
                    System.out.println("7");
                    break;
            }
        }
    } //main
}

```

Problem 12:

What is the output of the below program?

```

public class Problem12{
    public static void main(String args[]) {
        int arr[] = new int[5];

```

```

        for(int i = 0; i < 3; i++)
            arr[i] = i+1;

        System.out.println(arr[0] + arr[3]);
        System.out.println(arr[0] - 1 + arr[2] * 2);

        arr[1] = arr[0] + 10;

        System.out.println(arr[1] * arr[3]);
    } //main
} //class

```

Problem 13:

What is the output of the below program?

```

public class Problem13{
    public static void main(String args[]) {
        double arr[] = {5.0, 4.0, 2.1, 3.3, 4.4, 5.5};

        for(int i = 0; i < arr.length/2; i++)
            System.out.print(arr[i+1] + " ");

        arr[0] = arr[3];
        arr[2] = arr[0];

        System.out.println();
        for(int i = 0; i < arr.length; i++)
            System.out.print(arr[i] + "\n");
    } //main
} //class

```

Problem 14:

What is the output of the below program?

```

public class Problem14{
    public static void main(String args[]) {
        String word = "Apples";
        char arr[] = new char[word.length()];

        for(int i = 0; i < arr.length; i++)
            arr[i] = word.charAt(5 - i);

        System.out.print("Before: ");
        for(int i = 0; i < arr.length; i++)
            System.out.print(arr[i]);

        for(int i = 1; i < 5; i++)
            if(i % 2 == 0)
                arr[i] = Character.toUpperCase(arr[i]);
    }
}

```

```
        System.out.print("\nAfter: ");
        for(int i = 0; i < arr.length; i++)
            System.out.print(arr[i]);
    } //main
} //class
```

Additional Problems

DIRECTIONS: Answer each problem below to the best of your ability. Solutions are not provided in this book.

Problem 15:

Write a program to perform the following tasks:

1. Prompt the user to enter 9 integers (can be of any value).
2. Store each entry in a 1 dimensional array called *entries*.
3. Traverse the array and print *x* number of Z characters based on the entry in the array. If the element is 0 or negative, nothing will print.

Sample output:

Enter 9 integers: 1 2 3 4 0 5 6 7 -8 9

Z

ZZ

ZZZ

ZZZZ

ZZZZZ

ZZZZZZ

ZZZZZZZ

ZZZZZZZZZ

Problem 16:

Write a program to perform the following tasks:

1. The user enters 5 integer values between 0 and 7 from the command line.
2. If any of those values are not correct, the program terminates.
3. Store each element in an integer array called *entries*.
4. Given a char array below, print the corresponding character from the integer array (based on the entry from the user).

```
char letters[] = {'A', 'T', 'E', 'S', 'F', 'H', 'I', 'R'};
```

Sample output: (assumes arguments 5 6 7 2 4)
HIRES

Problem 17:

Write a program to perform the following tasks:

- Given a String array and an integer array (they are parallel to each other), print the names of each student who passed an exam. A passing grade is 70 or more.

```
String students[] = {"Sue", "John", "Lisa", "Frank", "Andrew", "Alex",
"James", "Christine"};
short scores[] = {71,68,90,100,55,76,89,23};
```

Sample output:

Sue
Lisa
Frank
Alex
James

Problem 18:

Write a program to perform the following tasks:

- Prompt the user to enter an integer *size* between 10 and 50.
- If the entered value is incorrect, the program terminates.
- Create a new 1 dimensional array of integers called *numbers* of the entered *size*.
- Prompt the user to enter *size* number of integers (do not error check for this program and assume correct entries).
- The program will determine what numbers were not entered by the user, based on the highest and lowest values entered in the array.

Sample output:

Enter an integer between 10 and 50: 10
Enter 10 integers: 1 2 4 5 1 9 2 12 3 14
The following were not entered: 6 7 8 10 11 13

Problem 19:

Write a program to perform the following tasks:

- Declare an array of type short named arr of size 10.
- Prompt the user to enter 10 integer values, positive or negative, no more than 100 in either direction.

3. The program replaces each value in the array with the sum of its previous values.
4. Print the values of the new array to the screen.

Sample output:

Enter 10 integer values:

3 5 2 7 8 10 12 1 13

3 8 10 17 25 35 47 48 61

Problem 20:

Given the below array, representing a transaction in a supermarket, write a program to perform the following tasks:

1. Calculate the total sum (subtotal) of the transaction. Print that value to the console.
2. Take the above value and add sales tax of 8.25%. Print this value to the console.
3. Prompt the user (or cashier in this case) for how much money the customer gave them. This value must be greater than or equal to the total in item 2. If it is not, the program terminates.
4. Print out the change that the cashier needs to give the customer.

```
double prices[] = {6.99, 4.95, 7.99, 7.99, 0.99, 12.49, 3.29, 4.39,  
5.97, 5.97, 11.99, 17.97};
```

Problem 21:

Write a program that will perform the following tasks:

1. Prompt the user to enter a single digit number between 1 and 9, inclusive.
2. Continue to ask the user for input until every number in that range was entered at least once.
3. If at any point a number entered is not in that range, ignore the input.
4. The program prints out the percentage of each digit that was entered.

Sample output:

1 was entered 21% of the time

2 was entered 12% of the time

3 was entered 3% of the time

4 was entered 5% of the time

5 was entered 8% of the time

etc.

TWO-DIMENSIONAL ARRAYS

A **two-dimensional** array in Java can be thought of as a table that has rows and columns. Below is the general declaration of a two-dimensional array:

```
type name[][] = new type[ rows ][ cols ];
```

Where in the above, *type* is the appropriate data type; *name* is the name of the array; *rows* is the number of rows; and *cols* is the number of columns.

Say we wanted to declare a two-dimensional array of integers with 5 rows and 6 columns:

```
int myarr[][] = new int[5][6];
```

The indices use the same principal as a one-dimensional array in the sense that they go from 0 to size-1 in each dimension. So here, the **row** indices would go from 0 to 4, and the **column** indices would go from 0 to 5.

Much like a one-dimensional array, you can declare an array and initialize it at the same time with the use of an initializer list. Again, there needs to be {} around the array you are initializing.

```
int heights[][] = {{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}};
```

This would declare and initialize an integer array of 2 rows and 5 columns, containing indices 0-1 for the rows and 0-4 for the columns.

```
boolean bools[][] = {{true, false}, {false, false}, {true, true}, {true, false}};
```

This would declare and initialize a boolean array of 4 rows and 2 columns, containing indices 0-3 for the rows and 0-1 for the columns.

```
String str[][] = {"One", "Two", "Three", "Four"}, {"One", "Two", "Three", "Four"}, {"One", "Two", "Three", "Four"};
```

This would declare and initialize a String array of 3 rows and 4 columns, containing indices 0-2 for the rows and 0-3 for the columns.

There are some shorthand ways of declaring and initializing two-dimensional arrays. These two declarations are the same:

```
int arr[][];  
int[][] arr;
```

Now, let's say you have more than one array you need to declare. These two below are the same:

```
int[][] a, b, c;  
int a[][], b[][], c[][];
```

EXAMPLE 6: Two-dimensional Array Basics

Here is short program making use of a two-dimensional array:

```
public class Example6{  
    public static void main(String args[]){  
        //declare 2d array with 2 rows and 3 columns  
        int myarr[][] = new int[2][3];  
  
        //loop through the array and insert values  
        for(int r = 0; r < 2; r++){  
            for(int c = 0; c < 3; c++){  
                myarr[r][c] = r*c+1;  
            }  
        }  
  
        //loop through array to print the values  
        for(int r = 0; r < 2; r++){  
            for(int c = 0; c < 3; c++){  
                System.out.print(myarr[r][c] + "\t");  
            }  
            System.out.println("");  
        }  
    } //main  
} //class
```

The above program will initialize the array to the values of $r*c+1$ and print out the values contained in those cells. The program shows how to use two-dimensional arrays.

The output from running the above program is:

1	1	1
1	2	3

EXAMPLE 7: Two-Dimensional Array Even Numbers

Below is a modification of example 3. This time, it will count the even and odd numbers in a two-dimensional array. The array is pre-filled with values of the row, plus the column loop counter.

```
public class Example7{
    public static void main(String args[]){
        //declare 2d array of int values with 5 rows and 5 columns
        int myarr[][] = new int[5][5];

        //loop to insert values into array
        for(int r = 0; r < 5; r++){
            for(int c = 0; c < 5; c++){
                myarr[r][c] = r+c;
            }
        }
        int evens = 0, odds = 0;

        //loop to count odd or even numbers
        for(int r = 0; r < 5; r++){
            for(int c = 0; c < 5; c++){
                if(myarr[r][c] % 2 == 0) evens++;
                else odds++;
            }
        }

        //print desired message
        System.out.println("There are " + evens + " numbers and "
            + odds + " odd numbers");

    } //main
} //class
```

The output from running the above program is:

There are 13 even numbers and 12 odd numbers.

Here is what the array looks like for reference purpose:

```
0 1 2 3 4  
1 2 3 4 5  
2 3 4 5 6  
3 4 5 6 7  
4 5 6 7 8
```

EXAMPLE 8: Multiplication Table

This program will calculate and print a multiplication table for the numbers 1-10.

```
public class Example8{  
    public static void main(String args[]){  
        //declare the array, ignoring index 0 in both row & column  
        int table[][] = new int[11][11];  
  
        //fill the values of the table  
        for(int r = 1; r <= 10; r++) {  
            for(int c = 1; c <= 10; c++) {  
                table[r][c] = r*c;  
            }  
        }  
  
        //print the table's top row  
        System.out.print("\t1\t2\t3\t4\t5\t6\t7\t8\t9\t10\n");  
  
        for(int r = 1; r <= 10; r++) {  
            //print the side number in the row  
            System.out.print(r + "\t");  
            for(int c = 1; c <= 10; c++) {  
                //print the actual number in the table  
                System.out.print(table[r][c] + "\t");  
            }  
            //go to next line  
            System.out.println();  
        }  
    } //main  
} //class
```

The output from running the above program will be:

	1	2	3	4	5	6	7	8	9	10
1	1	2	3	4	5	6	7	8	9	10
2	2	4	6	8	10	12	14	16	18	20
3	3	6	9	12	15	18	21	24	27	30
4	4	8	12	16	20	24	28	32	36	40
5	5	10	15	20	25	30	35	40	45	50
6	6	12	18	24	30	36	42	48	54	60
7	7	14	21	28	35	42	49	56	63	70
8	8	16	24	32	40	48	56	64	72	80
9	9	18	27	36	45	54	63	72	81	90
10	10	20	30	40	50	60	70	80	90	100

TWO-DIMENSIONAL ARRAY EXERCISES

DIRECTIONS: Answer each question below to the best of your ability. Solutions on page 630.

Problem 1:

Declare a two-dimensional array of the following:

- a) An array of integers called **matrix** of size 5 by 5.
- b) An array of floats called **scores** of size 10 by 4.
- c) An array of char called **picture** of size 12 by 12.
- d) An array of double precision numbers called **something** of size 4 by 6.
- e) An array of strings called **str** of size 10 by 100.

Problem 2:

Write a for loop that will perform each of the following tasks:

- a) Increment each element of a two-dimensional long array called **li** of size 10 by 5.
- b) Find the minimum value in a two-dimensional double array call **darr** of size 3 by 5.
- c) Find the count of the number of lowercase vowels in a two-dimensional char array called **letters** of size 10 by 4.
- d) Find the number of negative values in a two-dimensional array of type double named **vals** of size 6 by 6.
- e) Find the product of the positive values of a two-dimensional array of type int named **chart** of size 4 by 2.

Problem 3:

Write a while loop that will perform each of the following tasks:

- a) Increment each element of a two-dimensional double array called **dd** of size 10 by 5.
- b) Find the minimum value in a two-dimensional int array call **arr** of size 3 by 5.
- c) Count the number of upper case letters present in a two-dimensional array of type String called **words** of size 10 by 100.
- d) Find the lowest and highest values in a two-dimensional array of type short called **temperatures** of size 5 by 5.
- e) Increment each element by 2.667 in a two-dimensional array of floats called **numbers** of size 3 by 20.

Writing Programs

Problem 4:

Write a program to count the number of ones in any position of a number in a two-dimensional array. Display the total count for the entire array. An example is the number 12311. There are 3 ones in that number.

The array will look like this:

```
long nums[][] = {{18212, 91112, 2056, 3002, 60121}, {21121, 319922, 199921, 120092, 1014401}, {98874, 987321, 21245, 652, 2211}};
```

Problem 5:

Write a program to perform the following tasks:

1. Find the minimum value in each row of a two-dimensional array & print it out.
2. The sum of the even numbers in the entire array and print it out.
3. The total number of prime numbers in the array. Recall that a prime number is a number only divisible by 1 and itself.

The array will look like this:

```
int arr[][] = {{1,3,6,8},{7,5,3,4},{5,76,8,9},{78,6,4,11}};
```

Problem 6:

Write a program to perform the following tasks:

1. Find the maximum value in each row of a two-dimensional array & print it out.
2. Find the product of the odd numbers in the array and print it out.
3. Find the min value even number in each column and print it out.

The array will be the same as the above problem 5.

Problem 7:

Write a program to perform the following tasks:

1. Declare an integer array of size 3×3 called *array*.
2. Prompt the user to enter 9 integers, positive or negative.
3. The program outputs an X character based on each value of the array. If the value is 0 or negative, nothing prints. Separate each set of characters by a white space. There should be three rows of output to the console. A sample is below:

Sample of array:

```
int array[][] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
```

Sample output:

```
X XX XXX  
XXXX XXXXX XXXXXX  
XXXXXXXX XXXXXXXX XXXXXXXXXXX
```

Problem 8:

Write a program to perform the following tasks:

1. Declare a two-dimensional array of double precision values of size 2×5 called *entries*.
2. Prompt the user to enter 10 double precision values, positive or negative.
3. Store each element in the array.
4. The program then outputs the larger of two adjoining values in the array. A whitespace will separate the output.

Sample of the array:

```
double array[][] = {{5, 7, 8, 9, 2}, {12, 5, 1, 3, 3}};
```

Sample output:

```
Enter 10 double precision values: 5 7 8 9 2 12 5 1 3 3  
12.0 7.0 8.0 9.0 3.0
```

Problem 9:

Write a program to perform the following tasks:

1. Given the two-dimensional String array below, write a program that removes the given String in the first element of each row in the array from the remaining elements in the same row of each array.
2. Print the array to the screen before and after the change, not printing the first element in each row.

```
String array[][] = { {"a", "orange", "apple", "banana"),
                    {"s", "Mississippi", "Missouri", "Florida"},
                    {"3","123345323","331231","90876451"} };
```

Sample output:

Before:

```
orange apple banana  
Mississippi Missouri Florida  
123345323 90876541 331231
```

After:

```
ornge pple bnn  
Miiippi Miouri Florida  
12452 90876541 121
```

Additional Problems

DIRECTIONS: Answer each problem below to the best of your ability. Solutions are not provided in this book.

Problem 10:

Write a program to perform the following tasks:

1. Declare an integer array of 3 rows and 4 columns called *entries*.
2. Prompt the user to enter 12 integers, positive or negative.
3. The program then outputs the value of the product of each column, with each value being separated by a white space.
4. The program then output the value of the sum of each column, with each value being separated by a white space.
5. The program then outputs the value of the difference of each row, with each value being separated by a white space.

Problem 11:

Write a program to perform the following tasks:

1. Declare an array of double precision values of size 4 by 4 called *entries*.
2. Prompt the user to enter 16 double precision values, positive or negative.
3. Store the values entered into that array.
4. Print the values of the array to the console.
5. The program then swaps the values of each neighboring row of numbers.
6. Print the values of the array to the console.

Sample of the array:

```
double entries[][] = {{1,2,3,4}, {5,6,7,8},  
                      {9,10,11,12}, {13,14,15,16}};
```

Sample output:

```
1.0 2.0 3.0 4.0  
5.0 6.0 7.0 8.0  
9.0 10.0 11.0 12.0  
13.0 14.0 15.0 16.0
```

```
5.0 6.0 7.0 8.0  
1.0 2.0 3.0 4.0  
13.0 14.0 15.0 16.0  
9.0 10.0 11.0 12.0
```

Problem 12:

Write a program to perform the following tasks:

- Given the two-dimensional integer array below, write a program that will convert each row of the array into an integer.
- Print each number to the console.

```
int array[][] = { {9, 4, 5, 2, 5}, {8, 8, 8, 2, 0}, {12, 0, 2, 3, 4} };
```

Sample output:

```
94525  
88820  
120234
```

Problem 13:

Write a program to perform the following tasks:

- Prompt the user to enter a number between 3 and 10, inclusive.
- If the entry is not correct, prompt the user to re-enter until correct.
- Prompt the user to enter an uppercase letter.
- If the character entered is not uppercase or if it is not a letter, the program terminates.
- Create a two-dimensional array of that size of type char, with the rows and columns based on the entry.
- Print the array to the console.

7. If the number entered was odd, convert the characters of each even numbered column of the array to lowercase. If the number entered was even, convert the characters in each odd numbered column of the array to lowercase.
8. Print the array to the console.

Sample output:

Enter an integer between 3 and 10: 5

Enter a character: Z

```
ZZZZZ  
ZZZZZ  
ZZZZZ  
ZZZZZ  
ZZZZZ  
zZzZz  
zZzZz  
zZzZz  
zZzZz  
zZzZz
```

FOR EACH LOOPS

There is a variation of a for loop called a **foreach loop** (also called an **enhanced for loop**), that will iterate through arrays or collections of data (Collections are discussed in Chapter 19). The general declaration of a foreach loop is:

```
for (type var_name : array) {  
    //code  
}
```

Where in the above, *type* is the data type of *var_name*; *var_name* is a useful name of the iterator; and *array* is the actual array you are going to iterate through. It is important to note that the iterator data type **must match** the data type of the array.

Foreach loops iterate through each item in the given array or Collection. It stores each item in the variable you define (*var_name* in the above) in the foreach loop and executes the code in the body of the foreach loop. Foreach loops only work with **one-dimensional arrays**.

Let's see a code snippet below:

```
int arr[] = {1, 2, 3, 4, 5};  
for (int a : arr){  
    System.out.println(a);  
}
```

The above will simply print the elements of the array. Since the size of the array is 5, the loop will run 5 times.

Let's see another code snippet below:

```
int arr[] = {1, 2, 3, 4, 5};  
int sum = 0;  
for (int a : arr){  
    sum += a;  
}  
  
System.out.println(sum);
```

The above will calculate the sum of the array. Again, the loop runs 5 times. Let's see an example below using Strings.

```
String names[] = {"Alex", "James", "Matt", "Britney"};  
for (String name : names){  
    System.out.print(name.substring(2));  
}
```

This would produce the following output after its 4 runs:

exmestrtney

Let's convert some of this chapters exercises to foreach loops for better understanding.

EXAMPLE 9: *One-dimensional Sum*

This program will declare an array of type double of size 5, and fill it with values entered by the user. It then outputs the sum of the values of the array.

```
import java.util.Scanner;
public class Example9{
    public static void main(String args[]){
        Scanner s = new Scanner(System.in);

        double entries[] = new double[5];
        double sum = 0.0;
        int count = 0;

        //get the 5 numbers from the user
        while (count < 5) {
            System.out.println("Please enter a decimal: ");
            entries[count] = s.nextDouble();
            count++;
        }

        //find the sum of the numbers
        for(double i : entries)
            sum += i;

        System.out.println("The sum is: " + sum);
    }//main
} //class
```

A sample run of the above program is:

```
Please enter a decimal: 5
Please enter a decimal: 4
Please enter a decimal: 3.3
Please enter a decimal: -3.3
Please enter a decimal: 0
The sum is: 9.0
```

EXAMPLE 10: *Vertical Bar Graph with foreach Loops*

This program will simply ask the user for 10 numbers, and print 10 rows of stars. The size of a row is specified by the value entered by the user.

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

```

public class Example10{
    public static void main(String args[]){
        System.out.println("Please enter 10 numbers:");

        int arr[] = new int[10];

        Scanner s = new Scanner(System.in);

        //get the 10 values from the user
        for(int i = 0; i < 10; i++)
            arr[i] = s.nextInt();

        //go through each element of the array
        for(int j : arr) {
            //j will be the value from the array
            //if 0 or negative, no stars print
            for(int k = 0; k < j; k++)
                System.out.print("*");
            System.out.println();
        }
    } //main
} //class

```

A sample run of the above program may be:

Please enter 10 numbers:

5 4 3 2 1 0 1 2 3 4

**

*

*

**

EXAMPLE 11: *String Basics with foreach Loops*

Here is small program to show both a declaration and creation of Strings:

```

public class Example11{
    public static void main(String[] args){
        String str[] = new String[5];
        for(int i = 0; i < str.length; i++)
            str[i] = new String("Hello " + i);
        //perform a small change:
        str[0] = str[3];

        for(String s : str)
            System.out.println(s);
    } //main
} //class

```

The output from running the above program is:

```

Hello 3
Hello 1
Hello 2
Hello 3
Hello 4

```

Foreach loops will be used frequently when dealing with Collections (see Chapter 19).

ARRAYS & METHODS

Arrays can be used as parameters in a method. Here is what they may look like:

One-Dimensional parameter:

```

public static void something(double arr[]);
public static void something(int[] arr);

```

Two-Dimensional parameter:

```

public static void something(double[][] arr);
public static void something(short arr[][]);

```

Notice that for both a one-dimensional and two-dimensional array, you do not have to declare the size of it in the method. This is different from C++, since a two-dimensional array needs the second dimension declared.

When used in a method, all array parameters are passed *by reference*. Again, by the rules of passing by reference, if you change anything in the array within the method, you also change it elsewhere in the program.

Below are some examples showing the use of arrays (both one and two dimensional arrays) with methods.

EXAMPLE 12: *Arrays and Methods*

Let's see a small example of a one-dimensional array with a method:

```
public class Example12{
    public static void main(String args[]){
        int s[] = new int[5];

        for(int i = 0; i < 5; i++)
            s[i] = i*3;

        int sum = sumArray(s);

        System.out.println("sumArray: " + sum);
    } //main

    private static int sumArray(int arr[]){
        int sum = 0;

        for(int i = 0; i < arr.length; i++)
            sum += arr[i];

        return sum;
    } //sumArray
} //class
```

Here, we declare an array of size 5, and with the use of a for loop, we assign values to each cell. Then, we call the static method sumArray(), and pass the entire array to the method. What is returned is the sum of the values in the array.

The output is:

sumArray: 30

EXAMPLE 13: *Smallest Entry in a One-dimensional Array*

This example will find the smallest entry in a one-dimensional array:

```
public class Example13{
    private static int minArray(int arr[]) {
        int min = arr[0];
        for (int c = 1; c < arr.length; c++)
            if (arr[c] < min) min = arr[c];
        return min;
    }

    public static void main(String args[]) {
        int x[9] = {9,3,4,2,6,7,10,1,2};
        System.out.println(minArray(x));
    } //main
} //class
```

The above program will output 1.

EXAMPLE 14: *Smallest Entry in a Two-dimensional Array*

Conversely, this program will find the minimum entry in a two-dimensional array:

```
public class Example14{
    private static int minArray(int[][] arr, int rCap, int cCap) {
        int min = arr[0][0];
        for (int r = 0; r < rowCap; r++)
            for (int c = 0; c < colCap; c++)
                if (arr[r][c] < min) min = arr[r][c];
        return min;
    }
    public static void main(String args[]) {
        int x[3][5] = {{5,5,15,12,4},
                      {3,13,5,5,9},
                      {6,1,9,23,12}};
        System.out.println(minArray(x, 3, 5));
    } //main
} //class
```

This program will output 1, as well.

EXAMPLE 15: *Character Example*

Here is small program that works with two character arrays, as well as a method that will print the contents of a given array.

```
public class Example15{
    public static void main(String[] args){
        char[] arr = {'a','b','c','d','e'};
        char[] arr2 = new char[5];

        //first action
        for(int i = 0; i < 5; i++)
            arr2[i] = arr[4-i];

        printArr(arr2);

        //second action
        for(int i = 1; i < 3; i++)
            arr2[i] = arr[2-i];

        printArr(arr2);

        //third action
        arr[1] = arr[3] = arr[4];

        printArr(arr);

        //final action
        arr2[0] = arr[1];
        arr[2] = arr2[1];
        arr[4] = arr2[4];

        printArr(arr2);
    } //main

    //method to print the contents of an array
    private static void printArr(char[] a) {
        for(int i = 0; i < a.length; i++)
            System.out.print(a[i]);

        System.out.println();
    }
} //class
```

The output when running the above program is:

```
edcba  
ebaba  
aecee  
ebaba
```

Let's explore why. The first for loop will simply assign the values from the end of first array backwards. The first call to the method `printArr()` prints `arr2`, which is the first line of output above. At this point, both arrays look like this:

```
arr = {'a', 'b', 'c', 'd', 'e'};  
arr2 = {'e', 'd', 'c', 'b', 'a'};
```

The second for loop will assign, to `arr2[1]` the value of `arr[2-1]`, which in this case the character 'b'. The next iteration of the loop will assign, to `arr2[2]`, the value of `arr[2-2]`, which in this case is the character 'a'. It then makes the call to the print method for `arr2` and prints the second line of output above. At this point, both arrays look like this:

```
arr = {'a', 'b', 'c', 'd', 'e'};  
arr2 = {'e', 'b', 'a', 'b', 'a'};
```

The third action will assign the value of `arr[4]` to both `arr[3]` and `arr[1]`, which in this case is the character 'e'. It then makes a call to the method and prints the third line of output above. At this point, both arrays look like this:

```
arr = {'a', 'e', 'c', 'e', 'e'};  
arr2 = {'e', 'b', 'a', 'b', 'a'};
```

Finally, there are some assignments made to both arrays. The first will assign the value of `arr[1]` to `arr2[0]`, which in this case is the character 'e'. Next, the program assigns the value of `arr2[1]` to `arr[2]`, which in this case is the character 'b'. Finally, `arr[4]` is given the value of `arr2[4]`, which in this case is the character 'a'. The program then calls the `printArr()` method to print the contents of `arr2`, which is the final line of output to the program.

EXAMPLE 16: *Printing and Swapping Elements*

Another example deals with two methods, one for swapping array data and another for printing an array:

```
public class Example16{
```

```

private static void swapElements(int arr[], int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

private static void printArray(int arr[]) {
    for (int i = 0; i < arr.length; i++)
        System.out.print(arr[i] + " ");
    System.out.println();
}

public static void main(String args[]) {
    int x[5] = {3,4,5,6,7};

    printArray(x);
    swapElements(x, 2, 3);
    printArray(x);

} //main
} //class

```

The output from running the above program is:

```

3 4 5 6 7
3 4 6 5 7

```

EXAMPLE 17: *Grading the Class*

This program will ask for the names and grades of 5 students in a class. It then calculates what the class average was, in addition to the highest and lowest grade entered.

```

public class Example17{
    private static void swapElements(int arr[], int i, int j) {
        //Scanner
        Scanner s = new Scanner(System.in);

        //arrays
        String students[] = new String[5];
        double grades[] = new double[5];

        //prompt user for inputs
        for (int i = 0; i < 5; i++) {
            System.out.print("Enter student name and grade: ");
        }
    }
}

```

```

        students[i] = s.next();
        grades[i] = s.nextDouble();
    }

    //find the average for the class
    System.out.println("The average for the class was: "
        + classAverage(grades));

    int highest = 0, lowest = 0;

    //method returns the index of the highest and lowest
    //grade respectively
    highest = highestGrade(grades);
    lowest = lowestGrade(grades);

    //print the results of each
    System.out.println("The student with the best grade is "
        + students[highest] + " with a grade of "
        + grades[highest]);

    System.out.println("The student with the worse grade is "
        + students[lowest] + " with a grade of "
        + grades[lowest]);
} //main

//method to calculate the class average
private static double classAverage(double[] g) {
    double avg = 0.0;
    for(int i = 0; i < g.length; i++)
        avg += g[i];
    return (avg/5);
}

//method to find the index of the lowest grade in the class
private static int lowestGrade(double[] arr) {
    double min = arr[0];
    int index, lowest = 0;

    for (index = 1; index < arr.length; index++)
        if (arr[index] < min) {
            min = arr[index];
            lowest = index;
        }
    return lowest;
}

//method to find the index of the highest grade in the class
private static int highestGrade(double[] arr) {
    double max = arr[0];
    int index = 0, highest = 0;
}

```

```

        for (index = 1; index < arr.length; index++)
            if (arr[index] > max) {
                max = arr[index];
                highest = index;
            }
        return highest;
    }
} //class

```

A sample run of the above program can be:

```

Enter student name and grade: Alex 33
Enter student name and grade: Ryan 44
Enter student name and grade: Jack 55
Enter student name and grade: Jillian 98
Enter student name and grade: Freddy 73
The average for the class was: 60.6
The student with the best grade is Jillian with a grade of 98.0
The student with the worse grade is Alex with a grade of 33.0

```

ARRAYS & METHODS EXERCISES

DIRECTIONS: Answer each question below to the best of your ability. Solutions on page 636.

Problem 1:

Write header lines for each of the below methods. Do not write the full method, just the header line.

- A method called **findMax**, that will return the largest element of a one-dimensional double precision array.
- A method called **isThere**, that will return true if an element of a one-dimensional integer array argument is present.
- A method called **print**, that will print each element of a float array with a specified length.
- A method called **reverseElements**, that will reverse the elements of a one-dimensional array of type long.
- A method called **swapElements**, that will swap two elements in a one-dimensional double precision array argument.
- A method called **reverseChar**, that will reverse the characters of a one-dimensional char array argument.

- g) A method called **addArraySum**, that return the total sum of each element in two float two-dimensional arrays.
- h) A method called **findSum**, that will return the sum of the positive numbers in a one-dimensional array of integers.
- i) A method called **findPosPercent**, that will return the percentage of positive numbers in a one-dimensional array of type short.
- j) A method called **removeCapitals**, that will remove the capital letters of each element of a two-dimensional array of Strings of size 12 by 6.
- k) A method called **passingScore**, that will take two arguments, a one-dimensional integer array, and a integer, which will return the total number of passing scores in the array. The first argument is the array and the second argument represents the passing score.
- l) A method called **decreaseByX**, that will take two parameters, a one-dimensional integer array and an integer value. The method will decrease each element of the array by X (the second parameter).
- m) A method called **printCaps**, that will print the capital letters of a two-dimensional array of Strings of size 4 by 4.
- n) A method called **printEvens**, that will print only the even numbers in a one-dimensional array of short integers.
- o) A method called **printLarger**, that will print the larger of two adjoining elements in a one-dimensional array of double precision values. The array will always be even length.
- p) A method called **printASCII**, that will print the ASCII values of each element in a one-dimensional char array (ASCII chart is in Appendix A).
- q) A method called **replaceDigitWith**, that will replace each digit in each String element of a two-dimensional array of size 20 by 20 with a * character.
- r) A method called **invertNumbers**, which will invert each value in a two-dimensional array of short integers of size 10 by 3.
- s) A method called **findPositive2D**, that returns the count of the number of positive double precision values in a two-dimensional array of size 5 by 100.
- t) A method called **findProdNegative**, that returns the product of all negative values in a two-dimensional array of integers of size 50 by 10.

Problem 2:

Write each method *in full* from the above section. It is implied that all appropriate libraries are being used. For example, if you need something from the java.util library, you do not need to write import java.util.*.

Writing Programs

Problem 3:

Write a program to perform the following tasks:

1. Have the user enter an integer n , which will represent the size of a one-dimensional array of integers.
2. Declare the array and have the user enter the numbers.
3. With the use of methods, find and display the following:
 - a. The lowest number in the array.
 - b. The highest number in the array.
 - c. The total of the numbers in the array.
 - d. The avg of the numbers in the array.

Problem 4:

Write a program to perform the following tasks:

1. Using a two-dimensional array of integers of size 4 by 4, have the user enter the 16 numbers for the array.
2. With the use of methods, find and display the following:
 - a. The minimum value in each row.
 - b. The sum of the even numbers in the entire array.
 - c. The sum of all the numbers in the entire array.

Problem 5:

Write a program to perform the following tasks:

1. The user is prompted to enter 5 integers, positive or negative.
2. Each value is stored into a one-dimensional array of short integers called *entries*.
3. The program, with the use of a method called **printGrid**, will print a grid like the below, where each number in the array represents the column where an X should go. For all other spaces in the row, print a “-” symbol. If the entered value is 0 or negative, no X will print.

Sample output:

Enter 5 numbers: 1 3 3 4 2

```
X----  
--X--  
--X--  
---X-  
-X---
```

Problem 6:

Write a program to perform the following tasks:

1. The user is prompted to enter 8 single digit integers that are positive or 0.
2. Store each entry in a one-dimensional array called *entries*.
3. If any of the entries are incorrect, the program terminates.
4. The program, with the use of a method called **returnNum** that takes the integer array above, will convert the array into an integer value (example below).
5. With the use of a method called **returnRevNum** that takes the integer array above, print the returned value, which represents the conversion of the array into a reversed integer value (example below).

If array was (8, 7, 6, 5, 4, 3, 2, 1):

```
returnNum(array)           //returns 87654321  
returnRevNum(array)        //returns 12345678
```

Sample output:

Enter 8 numbers between 0 and 9: 8 7 6 5 4 3 2 1

87654321

12345678

Problem 7:

Write a program to perform the following tasks:

1. Ask the user to enter 10 integers, positive or negative.
2. Store each element in an array called *entries*.
3. With the use of a method called **smallestProd**, print the return value representing the smallest product of two adjoining elements in the array.
4. With the use of a method called **biggestDiff**, print the return value that represents the largest difference of two adjoining elements in the array.
5. With the use of a method called **biggestSum**, print the return value that represents the largest sum of two adjoining elements in the array.

Sample output:

Enter 10 integers: 5 1 6 2 4 2 3 1 9 2

Smallest product: 3

Biggest difference: 7

Biggest sum: 11

Output Tracing

Problem 8:

What is the output of each of the below programs that utilize recursive methods?

a)

```
public class A{
    public static void f( int a[], char b, int i){
        if(i > 3) return;

        System.out.println(a[i++] + "" + b);

        b+=2;
        a[0] = b;

        f(a, b, i);
    }
    public static void main(String args[]){
        int arr[]={1,2,3,4};

        char b = 65; //letter A;
        f(arr, b, 0);

        System.out.println(b);

        for(int i=0; i < 4; i++)
            System.out.print(arr[i] + " ");
    } //main
} //class
```

b)

```
public class B{
    public static void f( int a[], char b, int i){
        if(i >= 3) return;

        System.out.println(a[i++] + "" + b);
        b+=2;
        a[0] = b;

        f(a, b, i);
    }
    public static void main(String args[]){
        int arr[]={1,2,3,4};

        char b = 65; //letter A;
        f(arr, b, 0);

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

```
        for(int i=0; i < 4; i++)
            System.out.print(arr[i] + " ");
    } //main
} //class
```

c)

```
public class C{
    static void something(char[] ch, int x){
        if(x == 0)
            return;

        System.out.print( ch[x] + " " );
        something(ch, x-2);
    }
    static void somethingElse(char[] ch, int x){
        if(x == 0)
            return;

        System.out.print( ch[x] );
        somethingElse(ch, x-2);

        System.out.print( ch[x] );
    }
    public static void main(String args[]){
        char ch[] = "Welcome home!".toCharArray();

        something(ch, 10);
        System.out.println();

        somethingElse(ch, 10);
        System.out.println();
    } //main
} //class
```

d)

```
public class D{
    public static void q(int[] x, int i, char c){
        if(x[i] == 0)
            return;

        for(int j = 0; j < x[i]; j++)
            System.out.print(c);
        System.out.println();

        q(x, i+1, c);
    }
    public static void main (String[] args){
```

```
        int x[] = {4, 3, 6, 2, 4, 0, 1, 9, 2, 4};  
        q(x, 0, '*');  
    } //main  
}
```

e)

```
public class E{  
    public static void q(int[] x, int i, char c){  
        if(x[i] == 0)  
            return;  
  
        q(x, i+1, c);  
  
        for(int j = 0; j < x[i]; j++)  
            System.out.print(c);  
        System.out.println();  
    }  
    public static void main (String[] args){  
        int x[] = {4, 3, 6, 2, 4, 0, 1, 9, 2, 4};  
        q(x, 0, '*');  
    } //main  
}
```

f)

```
public class F{  
    public static void f(short[] x, int i){  
        if(i == 5 || x[i] == 0)  
            return;  
  
        System.out.println(x[i]);  
  
        f(x, i+1);  
  
        System.out.println(x[i]);  
    }  
    public static void main (String[] args){  
        short x[] = {4, 5, 2, 8, 0, 8, 4};  
  
        f(x, 0);  
    } //main  
}
```

Problem 9:

What is the output of the below program?

```
public class Problem9{  
    static void print(int[] x){  
        for(int i = 0; i < 10; ++i){  
            System.out.print(x[i] + " ");
```

```

        }
        System.out.println();
    }
    static void mix(int[] x){
        for(int i = 0; i < 10/2; i++)
            x[i] = x[10-i-1];
    }
    public static void main (String[] args){
        int x[] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};

        print(x);
        mix(x);
        print(x);
    }//main
}

```

Problem 10:

What is the output of the below program?

```

public class Problem10{
    static void print(int[] x){
        for(int i = 0; i < 10; ++i){
            System.out.print(x[i] + " ");
        }
        System.out.println();
    }

    static void mix(int[] x){
        int t = 0;
        for(int i = 0; i < 10/2; i++){
            t = x[i];
            x[i] = x[10-i-1];
            x[10-i-1] = t;
        }
    }

    public static void main (String[] args){
        int x[] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};

        print(x);
        mix(x);
        print(x);
    }//main
}

```

Problem 11:

What is the output of the below program?

```

public class Problem11{
    private static String getName(String arr[], int i){
        return arr[i];
    }
    private static String getMystery(String arr[], int i){
        String s = "";
        for(int x = 0; x < i; x++)
            s += (arr[x]).charAt(0);

        return s;
    }
    public static void main(String args[]){
        String str[] = {"Allan", "Peter", "Brian",
                        "Sally", "Susan"};

        for(int i = 0; i < 5; i++)
            System.out.println(getName(str, i) + " "
                               + getMystery(str, i+1));
    } //main
} //class

```

Problem 12:

What is the output of the below program?

```

public class Problem12{
    static void print(char c[], int x, int y){
        for(int i = x; i < y; i++){
            System.out.print( c[x] + " " );
        }
        System.out.println();
    }

    static void print2(long lg[], int x, int y){
        while(x < y){
            System.out.print( lg[x] + " " );
            x++;
        }
        System.out.println();
    }

    public static void main(String args[]){
        char x[] = {'H', 'e', 'l', 'l', 'o'};
        long lg[] = new long[12];

        for(int i = 0; i < 7; i++){
            lg[i] = i+3;
        }

        for(int i = ((int)lg[7]); i < lg[4]; i++)
            print2(lg, i, ((int)lg[4]));
    }
}

```

```

        for(int i = 0; i < 5; i++){
            print(x, i, 5);
            x[i] = x[i]++;
            print(x, i, 5);
        }
    } //main
} //class

```

Problem 13:

```

public class Problem13{
    static char[] mystery(char s, int x){
        String q = "";
        for(int i = x; i > 0; i--){
            q += ((char) s);
        }
        q += ((char) (x+65));
        return q.toCharArray();
    }

    public static void main(String args[]){
        char[] c, d, e;

        c = mystery('X', 12);
        d = mystery('Y', 24);
        e = mystery('Z', 6);
    } //main
} //class

```

1. What is contained in the character array *c*?
2. What is contained in the character array *d*?
3. What is contained in the character array *e*?

Program 14:

What is the output of the below program?

```

public class Problem14{
    static void thisIsCrazy(char z[], int len){
        for(int i = 0; i < len; i++){
            System.out.print(z[i] + " ");
        }
        System.out.println();
        for(int i = 0; i < len; i++)
            z[i]++;
    }
    public static void main(String args[]){

```

```

char[] x = new char[15];
char c = 65;

for(int i = 0; i < 15; i++)
    x[i] = (char)(c+i);

thisIsCrazy(x, 10);
thisIsCrazy(x, 15);

for(int i = 0; i < 15; i++)
    System.out.print(x[i] + " ");
} //main
} //class

```

Program 15:

```

public class Problem15{
    static counter1 = 0, counter2 = 0;

    public static int getSum(int a, int b, int c){
        ++counter1;

        return(a+b+c);
    } //getSum
    public static void printArr(int arr[], int len){
        for(int i = 0; i < len; i++){
            System.out.print(arr[i] + " ");
        }
        System.out.print("\n");
        ++counter2;
    } //printArr
    public static void main(String args[]){
        int a[] = {4,5,6,7,8};

        for(int i = 5; i >= 0; --i)
            printArr(a, --i);

        int sum = getSum(a[0], a[1], a[3]);

        System.out.println("Sum: " + sum);
    } //main
} //class

```

1. What is the value of counter1 at the end of the program?
2. What is the value of counter2 at the end of the program?
3. What is the output of the program?

Additional Problems

DIRECTIONS: Answer each problem below to the best of your ability. Solutions are not provided in this book.

Problem 16:

Write a program to perform the following tasks:

1. Prompt the user for 6 integers, positive or negative.
2. Store each element in a one-dimensional array called *entries*.
3. Write a method called **print**, that will print an integer array, which is its parameter.
4. With the use of a method called **multiFirst**, taking an array as its parameter, multiply each element of the array by the first element in the array (including the first element). Call the **print** method after the change within this method.

Sample output:

Enter 6 integers: 4 3 2 1 -5 0

16 12 8 4 -20 0

Problem 17:

Write a program to perform the following tasks:

1. Prompt the user for 5 integer values between 1 and 1000 (inclusive).
2. Store each element in an array called *nums*.
3. Write a method called **print**, which will print an integer array, that is its parameter.
4. Print the *nums* array after all the entries, via the **print** method.
5. With the use of a method called **squareEach**, update the *nums* array to reflect the square of each digit in each element of the array.
6. Print the *nums* array via the **print** method.

Sample output:

Enter 5 integers: 11 34 222 543

11 916 444 25169

Problem 18:

Write a program to perform the following tasks:

1. Prompt the user for 11 decimal values, positive or negative.
2. Store each value in a one-dimensional array called *nums*.

- With the use of a method called **series**, return a value representing the result of the series below:

entry1 + entry2 - entry3 + entry4 - entry5 ... -entry9 + entry10 - entry11

Problem 19:

Write a program to perform the following tasks:

- Prompt the user to enter an integer value x from 3 to 10, inclusive.
- If the entry is incorrect, prompt again until the user gets it right.
- Create a one-dimensional String array called *arr*, that's size is based on the entered value.
- Prompt the user to enter x String values.
- With the use of a method called **reverseVals**, reverse each element of the String array.
- With the use of a method called **printVals**, print the result of the above step.
- With the use of a method called **replaceVals**, replace each instance of an 'a' character in each element of the array with a ^.
- Call the **printVal** method again to print the result.

Problem 20:

Write a program to perform the following tasks:

- Declare a two-dimensional array of short integers of size 2 by 5.
- Prompt the user to enter 10 values. The 1st and 6th entry must be between 1 and 9.
- If the 1st and 6th value is incorrect, the program terminates.
- Store the values in the declared array as you go.
- With the use of a method called **multiRow2D**, write a program that will multiply each element in each row of the array by the first element of each row in the array.
- Print to the console the values after the conversion.

Sample output:

Array = {{4,2,2,2,2},
 {3,2,2,2,2}};

Enter 10 numbers: 4 2 2 2 3 2 2 2 2

16 8 8 8 8

9 6 6 6 6

Problem 21:

Given the below array, write a short program that will utilize a method called **firstDupe**, that will return a value representing the first occurrence of a duplicate value. The method returns the actual value, not the index of the occurrence.

Sample output:

```
int arr[] = {5, 3, 4, 4, 7, 8, 5};
```

4

Problem 22:

Given the below array, write a short program that will utilize a method called **firstUnique**, that will return a value representing the first occurrence of a unique value. The method returns the actual value, not the index of the occurrence.

Sample output:

```
int arr[] = {5, 3, 4, 4, 7, 8, 5};
```

3

Problem 23:

Given the below two-dimensional array, write a short program that utilizes a method called **matchingDigits**, that will return a value representing the number of matching digits between two values in the same position. The program will print 5 separate lines representing those values returned from the method.

Sample output:

```
int arr[][] = {{111, 111}, {123, 321}, {999762, 689733}, {4125, 4215}, {123456789, 123456789}};
```

3

1

2

2

9

Problem 24:

Write a program to perform the following tasks:

1. Ask the user for a positive integer x .

2. If the value entered is less than or equal to 0, the program terminates.
3. Create a new array of type String called *words* of size *x*.
4. The user enters *x* String values, with each being stored in the array.
5. Within a method called **isWrong**, go through each value of the array. If any entry is not of at least length 5, replace it with "WRONG."
6. With the use of a method called **print**, print each value of the array on a separate line to the console.

Sample output:

Enter a positive integer: 4

Fish

Lions

Bears

Cat

WRONG

Lions

Bears

WRONG

Problem 25:

Write a program to perform the following tasks:

1. Ask the user for a positive integer *x*.
2. If the value entered is less than or equal to 0, the program terminates.
3. Create a new array of type String called *words* of size *x*.
4. The user enters *x* String values, with each being stored in the array.
5. Within a method called **capEm**, go through each value of the array and capitalize each odd numbered character (based on index).
6. With the use of a method called **print**, print each value of the array on a separate line to the console.

Sample output:

Enter a positive integer: 5

FisH

lions

beaRs

Cat

dOgs

FIsH
lIoNs
bEaRs
CAt
dOgS

CHAPTER 11

Some Useful Libraries

This chapter gives an overview of Random numbers in Java, as well as the built-in Math library.

TOPICS

1. <i>Random Numbers</i>	250
2. <i>Random Number Seeds</i>	258
3. <i>Random Number Exercises</i>	261
4. <i>Math Library</i>	263
5. <i>BigInteger Class</i>	267

RANDOM NUMBERS

In Java, as well as any other programming language, you can generate a random number, or sequence of random numbers. In order to use this library, you must import it from the `java.util` library:

```
import java.util.Random;
```

Here is how to create a new Random object in Java:

```
Random name = new Random();
```

Where in the above, `name` is an appropriate name for the Random object. Once declared, you can begin to generate a random number by making use of an appropriate method within the object.

While it may appear that Java is indeed generating random numbers, it really isn't. It is generating what is called **pseudorandom numbers**. This is accomplished by the Random class taking an initial value (called a **seed**), that then generates another value based on the seed you give it. If no seed is defined, the default value is the computer's clock in milliseconds, which is of the type long.

To define a Random object with a seed, you would need to do this:

```
Random name = new Random();
name.setSeed (long value);
```

Where in the above, `name` is an appropriate name for the Random object. You then call the method within the Random object, and seed the value. The parameter of the `setSeed()` method is a long integer value.

You can also set the seed at the time of creating the new Random object. That can be accomplished like this:

```
Random name = new Random(100);
```

Generate Numbers

Contained within the Random object you create are a list of methods that will generate the pseudorandom numbers. They are defined below:

int nextInt()

This method will return a pseudorandom integer.

int nextInt(int range)

This method will return a pseudorandom integer in the range 0 to range-1. This is also very similar to array indices, as they go from 0 to size-1.

Say that you wanted to generate a random value from 1 to 10. This can be accomplished as:

```
int num = r.nextInt(10) + 1;
```

double nextDouble()

This method will return a pseudorandom double precision value.

float nextFloat()

This method will return a pseudorandom floating point value.

long nextLong()

This method will return a pseudorandom long integer.

boolean nextBoolean()

This method will return a pseudorandom boolean value.

Some example programs are below, showing some of the above methods and possible outputs.

EXAMPLE 1: Random Number Arrays

Here is an example program that will fill an array with random numbers:

```
import java.util.Random;
public class Example1{
```

```

public static void main(String args[]){
    Random r = new Random();
    int arr[] = new int[20];

    for(int i = 0; i < 20; i++){
        //random numbers from 1 to 10:
        arr[i] = r.nextInt(10) + 1;
    }

    for(int i = 0; i < 20; i++){
        System.out.print(arr[i] + " ");
    }
} //main
} //class

```

The program simply places random numbers from 1 to 10 inside the array called *arr*. It will output the values in the array in the second for loop. This is a simple program to see how random numbers work. Run the program multiple times to see different numbers appear.

EXAMPLE 2: Random Number Counting

This program will keep track of how many times a random number appeared.

```

import java.util.Random;
public class Example2{
    public static void main(String args[]){
        Random r = new Random();

        //allowing indices 1 to 25
        int arr[] = new int[26];

        for(int i = 0; i < 1000; i++){
            //random numbers from 1 to 25:
            arr[r.nextInt(25) + 1]++;
        }

        for(int i = 1; i <= 25; i++){
            System.out.println(i + " was generated " +
                arr[i] + " times.");
        }
    } //main
} //class

```

This program will keep track of how many times a random number was drawn. This will increment the appropriate place in the array. The generated number itself will be the index of the array. One possible output can be:

```
1 was generated: 33 times.  
2 was generated: 43 times.  
3 was generated: 44 times.  
4 was generated: 45 times.  
5 was generated: 39 times.  
6 was generated: 44 times.  
7 was generated: 40 times.  
8 was generated: 41 times.  
9 was generated: 43 times.  
10 was generated: 37 times.  
11 was generated: 38 times.  
12 was generated: 42 times.  
13 was generated: 37 times.  
14 was generated: 38 times.  
15 was generated: 42 times.  
16 was generated: 30 times.  
17 was generated: 37 times.  
18 was generated: 42 times.  
19 was generated: 38 times.  
20 was generated: 33 times.  
21 was generated: 52 times.  
22 was generated: 45 times.  
23 was generated: 51 times.  
24 was generated: 33 times.  
25 was generated: 33 times.
```

The output will be different each time the program is run.

EXAMPLE 3: *Random Stars*

This program simply generates a random amount of stars to the screen.

```
import java.util.Random;  
public class Example3{  
    public static void main(String args[]) {
```

```
Random r = new Random();
int num = 0;

for(int i = 0; i < 25; i++){
    //random numbers from 0 to 15:
    num = r.nextInt(16);
    for(int j = 0; j < num; j++)
        System.out.print("*");
    System.out.println();
}
} //main
} //class
```

The program will simply output rows of stars to the screen 25 times. However, you may not see 25 rows, since the inner loop will strictly be less than the generated number. If the random number is 0, the loop will not run and leave a blank row.

One possible output is this:

```
*****
**
*

*****
*****
*****
**
*****
*****



*****
***



*****



*****
*****
*****
***



*****



*****



***



**
```

Another possible output is this:

EXAMPLE 4: Rolling Dice

This program will roll a pair of dice 1000 times. It will calculate how many times a double was rolled, and print out each result to the user (*assume the user has a pair of fair dice, with the numbers 1 to 6 on the faces*).

```
import java.util.Random;
public class Example4{
    public static void main(String args[]) {
        Random d1, d2;

        //initialize new Random objects representing
        //two dice
        d1 = new Random();
        d2 = new Random();

        //array to track the doubles, allows for using index 1 to 6
        int pairs[] = new int[7];
        int d1_roll = 0, d2_roll = 0;

        //roll the pair 1000 times
        for(int i = 0; i < 1000; i++) {
            //roll the pair of dice (numbers 1 to 6)
            d1_roll = d1.nextInt(6) + 1;
            d2_roll = d2.nextInt(6) + 1;
            if(d1_roll == d2_roll) {
                pairs[d1_roll]++;
            }
        }
    }
}
```

```

        d1_roll = d1.nextInt(6) + 1;
        d2_roll = d2.nextInt(6) + 1;

        //same number was rolled
        if(d1_roll == d2_roll)
            pairs[d1_roll]++;
    }

    //print the results to the user
    for(int i = 1; i <= 6; i++)
        System.out.println("Pairs of " + i + "s were rolled "
                           + pairs[i] + " times.");
} //main
} //class

```

Running the above program can produce an output of:

Pairs of 1s were rolled 25 times.
 Pairs of 2s were rolled 27 times.
 Pairs of 3s were rolled 22 times.
 Pairs of 4s were rolled 36 times.
 Pairs of 5s were rolled 24 times.
 Pairs of 6s were rolled 33 times.

Or this:

Pairs of 1s were rolled 34 times.
 Pairs of 2s were rolled 22 times.
 Pairs of 3s were rolled 21 times.
 Pairs of 4s were rolled 26 times.
 Pairs of 5s were rolled 41 times.
 Pairs of 6s were rolled 30 times.

The results will be different each trial. When we get to dealing with file input/output in Chapter 16, we will further expand on the above program.

EXAMPLE 5: *One-dimensional Card Shuffler*

This program will deal a hand of 5 cards to the user. It also checks whether or not the card has already been dealt from the deck.

```

import java.util.Random;
public class Example5{
    public static void main(String args[]){
        Random r = new Random();

        //declare the arrays for the number and suit of the cards
        String number[] =
        {"2","3","4","5","6","7","8","9","10","J","Q","K","A"};
        String suit[] = {"Hearts","Diamonds","Clubs","Spades"};

        //array to handle the check whether the card has been
        //dealt from the deck. The rows indicate the suits
        //while the columns indicate the cards
        int cards[][] = new int[4][13];

        for(int i = 0; i < 5; i++){
            int col = r.nextInt(13); //next card
            int row = r.nextInt(4); //next suit

            //in case of a duplicate:
            while(cards[row][col] == 1){
                row = r.nextInt(4); //next suit
                col = r.nextInt(13); //next card
            }

            cards[row][col]++;
            //print the card to the screen
            System.out.println(number[col] + " of "
                + suit[row]);
        }

        //print out the array for a check:
        for(int row = 0; row < 4; row++){
            for(int col = 0; col < 13; col++){
                System.out.print(cards[row][col] + " ");
            }
            System.out.println();
        }
    } //main
} //class

```

A couple of new things have occurred here. Firstly, the two-dimensional array called *cards* will be used to check if that card of the suit and number have already been dealt. The rows of the cards array will represent the suits of the deck, while the columns will represent the number. We will use only the numbers 1 and 0, where 1 represents a card that has already been dealt, and 0 means it has not. When we check for the card, we will

use a while loop inside the for loop. The while logic has to be when the `card[row][col]` is 1, signifying that the card has already been dealt. If this is true, the while loop will randomly pick another pair of numbers and try again.

Finally, the two-dimensional array is printed out to the console to check whether or not the cards we dealt correctly. Here is one possible output from running the above program:

```
J of Spades  
6 of Clubs  
9 of Clubs  
5 of Hearts  
Q of Diamonds  
0 0 0 1 0 0 0 0 0 0 0 0  
0 0 0 0 0 0 0 0 0 1 0 0  
0 0 0 0 1 0 0 1 0 0 0 0  
0 0 0 0 0 0 0 0 1 0 0 0
```

Notice that there are no duplicates. There will be no duplicates each time the program is run.

RANDOM NUMBER SEEDS

As stated, the default value of the seed is the system's time in milliseconds. But what if we defined our own seed value?

EXAMPLE 6: Random Number Seeds

Here, we will generate 10 random numbers using a defined seed for the Random object.

```
import java.util.Random;  
public class Example6{  
    public static void main(String args[]){  
        //set a specific seed for the RNG  
        Random r = new Random(1900);  
  
        //print 10 random numbers  
        for(int i = 0; i < 10; i++)  
            System.out.println(r.nextInt());  
    } //main
```

```
} //class
```

Running the program above, we may see the following set of numbers:

```
-1689516050  
1065834624  
-1854745320  
-2037429205  
902967256  
-1539429153  
1525273823  
943015176  
1932264495  
-2069858995
```

Run the program again with the same seed value:

```
-1689516050  
1065834624  
-1854745320  
-2037429205  
902967256  
-1539429153  
1525273823  
943015176  
1932264495  
-2069858995
```

The same sequence of numbers is generated.

EXAMPLE 7: Rolling Dice with a Seed Value

This program is a modification of Example 4 above. It will roll a pair of fair dice 1000 times and calculate how many times a double was rolled. It then prints each result to the console. The change in this program is the use of two defined seeds for the Random objects.

```

import java.util.Random;
public class Example7{
    public static void main(String args[]) {
        Random d1, d2;

        //initialize new Random objects representing
        //two dice
        d1 = new Random(100);
        d2 = new Random(200);

        //array to track the doubles, allows for using index 1 to 6
        int pairs[] = new int[7];
        int d1_roll = 0, d2_roll = 0;

        //roll the pair 1000 times
        for(int i = 0; i < 1000; i++) {
            //roll the pair of dice (numbers 1 to 6)
            d1_roll = d1.nextInt(6) + 1;
            d2_roll = d2.nextInt(6) + 1;

            //same number was rolled
            if(d1_roll == d2_roll)
                pairs[d1_roll]++;
        }

        //print the results to the user
        for(int i = 1; i <= 6; i++)
            System.out.println("Pairs of " + i + "s were rolled "
                               + pairs[i] + " times.");
    } //main
} //class

```

Running the above program can produce an output of:

Pairs of 1s were rolled 32 times.
 Pairs of 2s were rolled 33 times.
 Pairs of 3s were rolled 19 times.
 Pairs of 4s were rolled 29 times.
 Pairs of 5s were rolled 33 times.
 Pairs of 6s were rolled 27 times.

Rerunning the above program a million times will produce the same output as the above, due to the defined seed values.

RANDOM NUMBER EXERCISES

DIRECTIONS: Answer each question below to the best of your ability. Solutions on page 648.

Problem 1:

Write a program that will randomly draw a 3 digit lottery number (000 to 999).

Problem 2:

Write a program that will randomly draw a 4 digit lottery number (0000 to 9999).

Problem 3:

Write a program that will randomly draw 10 even numbers in the range 1 to 50.

Problem 4:

Write a program that will randomly draw 10 odd numbers in the range 1 to 100.

Problem 5:

Write a program that will count the number of randomly drawn numbers from 0 to 1000, that are evenly divisible by both 2 & 5. Perform this task when drawing 50 random numbers. Display the number only when the number is divisible by both 2 & 5.

Problem 6:

Write a program that will display numbers ending in 7, which are randomly drawn in the range 1 to 777. Perform this task when drawing 100 random numbers. Display the number only when it ends in 7.

Problem 7:

Using random numbers, generate a one-dimensional array of short integers of size 20, whose values are between 1 and 50. Find the mean (average) of the generated values.

Additional Problems

DIRECTIONS: Answer each problem below to the best of your ability. Solutions are not provided in this book.

Problem 8:

Write a program that will perform the following tasks:

1. Fill an array named *rands* of size 5 with random, positive integer values between 1 and 100000.

- With the use of a method called **sumOdd**, print out the returned value of the method, which represents the sum of the odd numbers in the array.

Problem 9:

Write a program that will perform the following tasks:

- Declare a two dimensional array of type char named *vals* of size 4 by 3.
- With the use of a method called **randFill**, fill the array with random lowercase letters (hint: lowercase a has an ASCII code of 97).
- With the use of a method called **randOut**, print out 7 random char values from the array.

Problem 10:

Write a program that will perform the following tasks:

- Declare a two-dimensional array of type short named *board*, of size 8 by 8.
- With the use of a method called **randFill**, fill the array with random 1s or 0s.
- Prompt the user to enter a coordinate value *x, y*. Each value must be between 0 and 7 inclusive.
- If either of the values are incorrect, the program terminates.
- If the value in the array at the given coordinate is a 1, print to a JOptionPane "HIT!" Otherwise, print "MISS" to the console.

Problem 11:

Write a program that performs the following tasks:

- Generates a random, positive 3 digit integer called *num*.
- Continue to generate a random 3 digit integer called *gen* for 1000 times.
- Keep track of the number of times the initial random number *num* is generated.
- Print the above value in step 3 to the console.

Problem 12:

Write a program that performs the following tasks:

- Randomly generate two values, *h1* (between 0 and 23 inclusive) and *m1* (between 0 and 59 inclusive), representing a time of day in hours and minutes. Print those values to the console.
- Randomly generate another two values, *h2* and *m2*, representing another time of day. Print those values to the console.
- The program prints out the length of time, in hours and minutes, that elapsed between those two times.

Sample output:

1:23

14:22

12:59

MATH LIBRARY

Java provides its programmers with a built-in Math library. It is accessible via Math.methodName() and does not require the import of any particular Java library. We have written many programs so far that could have made use of the Math library to perform our desired tasks.

Below is a list of the various methods associated with this library.

```
static double abs(double d)
static float abs(float f)
static int abs(int n)
static long abs(long l)
>Returns the absolute value of its argument.

static double cbrt(double a)
>Returns the cube root of a double value.

static double ceil(double a)
>Returns the smallest double precision value that is not less than the
argument and is equal to a mathematical integer.

static double cos(double a)
>Returns the trigonometric cosine of an angle.

static double cosh(double a)
>Returns the hyperbolic cosine of an angle.

static double exp(double a)
>Returns Euler's number e raised to the power of a double value.

static double floor(double a)
>Returns the largest double precision value that is not greater than the
argument and is equal to a mathematical integer.

static double hypot(double x, double y)
>Returns the sqrt(x2 + y2).

static double log(double a)
>Returns the natural logarithm (base e) of a double value.
```

static double log10(double a)

Returns the natural logarithm (base 10) of a double value.

static double max(double a, double b)

static float max(float a, float b)

static int max(int a, int b)

static long max(long a, long b)

Returns the greater of two values of the appropriate data type.

static double min(double a, double b)

static float min(float a, float b)

static int min(int a, int b)

static long min(long a, long b)

Returns the smaller of two values of the appropriate data type.

static double nextDown(double a)

Returns the floating-point value adjacent to the parameter in the direction of negative infinity.

static double nextUp(double a)

Returns the floating-point value adjacent to the parameter in the direction of positive infinity.

static double pow(double a, double b)

Returns the value of the first argument raised to the power of the second argument.

static double random()

Returns a double precision value with a positive sign, greater than or equal to 0.0 and less than 1.0.

static long round(double a)

Returns the closest long to the argument.

static int round(float a)

Returns the closest int to the argument.

static double sin(double a)

Returns the trigonometric sine of an angle.

static double sinh(double a)

Returns the hyperbolic sine of an angle.

static double sqrt(double a)

Returns the correctly rounded positive square root of a double value.

static double tan(double a)

Returns the trigonometric tangent of an angle.

static double tanh(double a)

Returns the hyperbolic tangent of an angle.

```
static double toDegrees(double angrad)
```

Returns the conversion of an angle measured in radians to an approximately equivalent angle measured in degrees.

```
static double toRadians(double angdeg)
```

Returns the conversion of an angle measured in degrees to an approximately equivalent angle measured in radians.

EXAMPLE 8: *Math Library Highlights*

This program asks the user for two numbers, and performs various actions on them via the Math library.

```
import java.util.Scanner;
public class Example8{
    public static void main(String args[]){
        Scanner s = new Scanner(System.in);
        double n1, n2;

        //prompt and get entries
        System.out.println("Enter 2 numbers: ");
        n1 = s.nextDouble();
        n2 = s.nextDouble();

        //show the various Math methods
        System.out.println("Absolute 1: " + Math.abs(n1));
        System.out.println("Absolute 2: " + Math.abs(n2));

        System.out.println("Ceiling 1: " + Math.ceil(n1));
        System.out.println("Ceiling 2: " + Math.ceil(n2));

        System.out.println("Floor 1: " + Math.floor(n1));
        System.out.println("Floor 2: " + Math.floor(n2));

        System.out.println("Max: " + Math.max(n1, n2));
        System.out.println("Min: " + Math.min(n1, n2));

        System.out.println("Next up 1: " + Math.nextUp(n1));
        System.out.println("Next up 2: " + Math.nextUp(n2));

        System.out.println("Next down 1: " + Math.nextDown(n1));
        System.out.println("Next down 2: " + Math.nextDown(n2));

        System.out.println("n1 ^ n2: " + Math.pow(n1, n2));

        System.out.println("Round n1: " + Math.round(n1));
```

```

        System.out.println("Round n2: " + Math.round(n2));

        System.out.println("Square root of n1: " + Math.sqrt(n1));
        System.out.println("Square root of n2: " + Math.sqrt(n2));

        System.out.println("Cube root of n1: " + Math.cbrt(n1));
        System.out.println("Cube root of n2: " + Math.cbrt(n2));
    } //main
} //class

```

A sample output of the above program can be:

```

Enter 2 numbers:
4.5 5.5
Absolute 1: 4.5
Absolute 2: 5.5
Ceiling 1: 5.0
Ceiling 2: 6.0
Floor 1: 4.0
Floor 2: 5.0
Max: 5.5
Min: 4.5
Next up 1: 4.5000000000000001
Next up 2: 5.5000000000000001
Next down 1: 4.499999999999999
Next down 2: 5.499999999999999
n1 ^ n2: 3914.4326552141665
Round n1: 5
Round n2: 6
Square root of n1: 2.1213203435596424
Square root of n2: 2.345207879911715
Cube root of n1: 1.6509636244473134
Cube root of n2: 1.7651741676630315

```

EXAMPLE 9: Trigonometry Highlights

This program performs various trigonometric calculations via the Math library.

```

public class Example9{
    public static void main(String args[]) {

```

```

//variables
double angle1 = 45, angle2 = 90, angle3 = 180;

//convert the above to radians
angle1 = Math.toRadians(angle1);
angle2 = Math.toRadians(angle2);
angle3 = Math.toRadians(angle3);

//show the various Math trigonometry methods
System.out.println("Sin of angle1: " + Math.sin(angle1));
System.out.println("Sin of angle2: " + Math.sin(angle2));
System.out.println("Sin of angle3: " + Math.sin(angle3));

System.out.println("Cos of angle1: " + Math.cos(angle1));
System.out.println("Cos of angle2: " + Math.cos(angle2));
System.out.println("Cos of angle3: " + Math.cos(angle3));

System.out.println("Tan of angle1: " + Math.tan(angle1));
System.out.println("Tan of angle2: " + Math.tan(angle2));
System.out.println("Tan of angle3: " + Math.tan(angle3));
} //main
} //class

```

A sample output of the above program will be:

```

Sin of angle1: 0.7071067811865475
Sin of angle2: 1.0
Sin of angle3: 1.2246467991473532E-16
Cos of angle1: 0.7071067811865476
Cos of angle2: 6.123233995736766E-17
Cos of angle3: -1.0
Tan of angle1: 0.9999999999999999
Tan of angle2: 1.633123935319537E16
Tan of angle3: -1.2246467991473532E-16

```

BIGINTEGER CLASS

There are times, when trying to solve certain math problems, that you deal with numbers that are larger than a *long* data type. Java has a BigInteger class that can handle such tasks.

To use the BigInteger class, you must import it from the java.math library:

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

To declare a BigInteger object:

```
BigInteger name = new BigInteger( String val );
```

Where in the above, **name** is a useful name of the object. Notice that the constructor for the BigInteger object contains a String argument, which represents the number. Such examples are:

```
BigInteger num = new BigInteger("2");
BigInteger num2 = new BigInteger("222");
BigInteger num3 = new BigInteger("2032958492302943854865");
BigInteger num4 = new
BigInteger("22312313423234557456434135244573456341246457354625352573457
635244573456234583457576");
```

Yes, you saw that last one correctly! This class can handle numbers that large, or even larger!

Member Methods

Many of the below methods, that are contained in the BigInteger class, will be similar to the Math.XXX methods, only here they mainly use or return BigInteger types.

BigInteger abs()

Returns the absolute value of a BigInteger.

BigInteger add(BigInteger val)

Returns the addition of two BigInteger objects (*this + val*).

BigInteger divide(BigInteger val)

Returns the division of two BigInteger objects (*this / val*).

BigInteger max(BigInteger val)

Returns the larger of two BigInteger objects.

BigInteger min(BigInteger val)

Returns the smaller of two BigInteger objects.

BigInteger mod(BigInteger val)

Returns the modular division of two BigInteger objects (this % val).

BigInteger multiply(BigInteger val)

Returns the product of two BigInteger objects (this * val).

BigInteger negate()

Returns the negation of a BigInteger object.

BigInteger pow(int exponent)

Returns a BigInteger object whose value is this ^ exponent (note, the argument is an int value)

BigInteger subtract(BigInteger val)

Returns the subtraction of two BigInteger objects (this - val).

double doubleValue()

This method converts a BigInteger to a double value.

float floatValue()

This method converts a BigInteger to a float value.

int intValue()

This method converts a BigInteger to an int value.

long longValue()

This method converts a BigInteger to a long value.

String toString()

Returns the String representation of a BigInteger object.

boolean equals(Object x)

Returns true if two BigInteger objects are exactly equal and false otherwise. The argument uses the generic Object type.

Let's see an example, showcasing most of the above methods.

EXAMPLE 10: BigInteger Highlights

This program performs various calculations with the use of the BigInteger class.

```
import java.math.BigInteger;
public class Example10{
```

```

public static void main(String args[]){
    BigInteger b1, b2;

    b1 = new BigInteger("15");
    b2 = new BigInteger("100");

    //show various methods using the BigInteger's above
    System.out.println("Sum (b1 + b2): " + b1.add(b2));
    System.out.println("Diff (b1 - b2): " + b1.subtract(b2));
    System.out.println("Product (b1 * b2): "
        + b1.multiply(b2));
    System.out.println("Quotient (b1 / b2): " + b1.divide(b2));
    System.out.println("Mod Division (b1 % b2): "
        + b1.mod(b2));
    System.out.println("Max: " + b1.max(b2));
    System.out.println("Min: " + b1.min(b2));
    System.out.println("Negate b1: " + b1.negate());
    System.out.println("Negate b2: " + b2.negate());
    System.out.println("Power (b1 ^ b2): "
        + b1.pow(b2.intValue()));

} //main
} //class

```

The output of the above program will be:

```

Sum (b1 + b2): 115
Diff (b1 - b2): -85
Product (b1 * b2): 1500
Quotient (b1 / b2): 0
Mod Division (b1 % b2): 15
Max: 100
Min: 15
Negate b1: -15
Negate b2: -100
Power (b1 ^ b2):
40656117753521523739727970756704167101038789063237976342905176987875638
31961701377171181093217455781996250152587890625

```

Yep, you read the last number above correctly!! This class has the power to generate very, very large numbers (Hint: this class will be quite useful for some of our programming challenges (see Appendix D)).

The BigInteger class can also help us find large prime numbers with some useful methods. Recall that a prime number is a number that is only divisible by 1 and itself, otherwise it is a composite number.

boolean isProbablePrime(int certainty)

Returns true if the BigInteger object is most likely a prime number and false otherwise (meaning it's a composite number). If the certainty parameter is ≤ 0 , true is always returned. If the call returns true, the probability that this BigInteger is prime exceeds $(1 - (1/2^{certainty}))$.

BigInteger nextProbablePrime()

Returns the next probable prime number that is larger than the current BigInteger value.

For the isProbablePrime() method, let's see some actual math to prove the validity of the method. First, if you gave a certainty value of 1:

$$\text{Result} = (1 - (1/2^1))$$

$$\text{Result} = (1 - (1/2))$$

$$\text{Result} = .5 \text{ (or } 50\%)$$

What if you gave the certainty parameter a value of 2?

$$\text{Result} = (1 - (1/2^2))$$

$$\text{Result} = (1 - (1/4))$$

$$\text{Result} = .75 \text{ (or } 75\%)$$

Let's see a value of 5:

$$\text{Result} = (1 - (1/2^5))$$

$$\text{Result} = (1 - (1/32))$$

$$\text{Result} = .96875 \text{ (or } 96.875\%)$$

The higher the value of certainty, the more likely you want the number to be prime.

Let's see a short example.

EXAMPLE 11: *BigInteger Primes*

This program will output some likely prime numbers, as well as check if a given number is prime.

```
import java.math.BigInteger;
public class Example11{
    public static void main(String args[]) {
        BigInteger b1, b2, b3;

        b1 = new BigInteger("37"); //known prime
        b2 = new BigInteger("99"); //not prime

        //print results by checking if numbers are likely primes
        System.out.println("B1 is prime? "
            + b1.isProbablePrime(1));
        System.out.println("B2 is prime? "
            + b2.isProbablePrime(10));

        //perform some math and create a new BigInteger
        b3 = new BigInteger( b1.add(b2).toString() );

        //show the results of the addition
        System.out.println("B3 = " + b3);
        //use method with more certainty
        System.out.println("B3 is prime? "
            + b3.isProbablePrime(100));

        //find the next likely prime after the value of b3
        System.out.println("Likely next prime is: "
            + b3.nextProbablePrime());
    } //main
} //class
```

The output when running the above program is:

```
B1 is prime? true
B2 is prime? false
B3 = 136
B3 is prime? false
Likely next prime is: 137
```

All true! 37 is indeed a prime number. 99 is most definitely not prime, as 3 can divide 99, 11 times. The addition of 99 & 37 results in 136, which is not a prime (all even numbers are not primes, except for 2). And the next prime number after 136 is 137.

We used small numbers to show the validity of the methods. But now let's change to larger numbers.

The output when $b1 = 6010343$ and $b2 = 99999123$ is:

```
B1 is prime? true  
B2 is prime? false  
B3 = 106009466  
B3 is prime? false  
Likely next prime is: 106009511
```

Keep experimenting with larger numbers than the above. It gets fun! (hint: this, too, will help with some of our programming challenges!)

Factorial Method Revisited

In Chapter 9 on Recursion, we dealt with the factorial method. Recall the below iterative solution:

```
public static long fact( long n ){  
    long ans = 1;  
    for(long i = n; i > 1; i--)  
        ans *= i;  
    return ans;  
}
```

Also recall the method uses type *long*, which is helpful, but still does not handle large numbers for this problem. That is where the class *BigInteger* comes in handy!

The method can be rewritten to the below:

```
public static BigInteger factorial(long n) {  
    BigInteger ans = new BigInteger("1");  
    String num = "";  
  
    for(long i = n; i > 1; i--) {  
        num = i + "";  
        ans = ans.multiply(new BigInteger(num));  
    }  
    return ans;
```

```
}
```

Let's see a program that uses both the iterative and BigInteger methods.

EXAMPLE 12: *BigInteger Factorial*

```
import java.math.BigInteger;
public class Example12{
    public static BigInteger factorial(long n) {
        BigInteger ans = new BigInteger("1");
        String num = "";

        for(long i = n; i > 1; i--) {
            num = i + "";
            ans = ans.multiply(new BigInteger(num));
        }
        return ans;
    }
    public static long factorialOld(long n){
        long ans = 1;
        for(long i = n; i > 1; i--)
            ans *= i;
        return ans;
    }
    public static void main(String args[]) {
        //print the BigInteger factorial of 20
        System.out.println("BigInt fact(20): " + factorial(20));

        //try to print the long factorial of 20
        System.out.println("Long fact(20): " + factorialOld(20));

        //print the BigInteger factorial of 20
        System.out.println("BigInt fact(50): " + factorial(50));

        //try to print the long factorial of 20
        System.out.println("Long fact(50): " + factorialOld(50));

        //print the BigInteger factorial of 20
        System.out.println("BigInt fact(100): " + factorial(100));

        //try to print the long factorial of 20
        System.out.println("Long fact(100): " + factorialOld(100));
    } //main
} //class
```

The output from the above program will be:

```
BigInt fact(20): 2432902008176640000
Long fact(20): 2432902008176640000
BigInt fact(50):
304140932017133780436126081660647688443776415689605120000000000000
Long fact(50): -3258495067890909184
BigInt fact(100):
93326215443944152681699238856266700490715968264381621468592963895217599
993229915608941463976156518286253697920827223758251185210916864000000000
00000000000000000
Long fact(100): 0
```

As is seen in the above, after the factorial of 20, the long method for factorial() does not produce the correct number, as it has overflowed the data type. However, BigInteger will produce the correct values for the factorial of 50 and 100.

Again, feel free to experiment with changing the values!

Fibonacci Method Revisited

Also in Chapter 9, we learned about the Fibonacci sequence. Recall the iterative method:

```
public static long fib( long n ){
    long n1 = 1, n2 = 1;
    for(int i = 3; i <= n; i++){
        long c = n1 + n2; //sum of previous 2
        n1 = n2; //old n2
        n2 = c; //new number is the sum
    }
    return n2;
}
```

Again, using type long, at some point the numbers get quite large. BigInteger will help! Let's see the revised method:

```
public static BigInteger fib(long n) {
    BigInteger n1 = new BigInteger("1");
```

```

BigInteger n2 = new BigInteger("1");
BigInteger sum;

for(long i = 3; i <= n; i++) {
    //String representation
    sum = new BigInteger( (n1.add(n2)).toString() );
    n1 = n2; //old n2
    n2 = sum; //new number is sum
}
return n2; //returns Nth number
}

```

Let's see an example, searching for some Fibonacci numbers.

EXAMPLE 13: *BigInteger Fibonacci*

```

import java.math.BigInteger;
public class Example13{
    public static BigInteger fib(long n) {
        BigInteger n1 = new BigInteger("1");
        BigInteger n2 = new BigInteger("1");
        BigInteger sum;

        for(long i = 3; i <= n; i++) {
            //String representation
            sum = new BigInteger( (n1.add(n2)).toString() );
            n1 = n2; //old n2
            n2 = sum; //new number is sum
        }
        return n2; //returns Nth Fibonacci number
    }
    public static long fibOld( long n ){
        long n1 = 1, n2 = 1;

        for(int i = 3; i <= n; i++){
            long c = n1 + n2; //sum of previous 2
            n1 = n2; //old n2
            n2 = c; //new number is the sum
        }
        return n2; //returns Nth Fibonacci number
    }
    public static void main(String args[]) {
        //print 10th Fibonacci number using BigInteger method
        System.out.println("BigInt fib(10): " + fib(10));

        //print 10th Fibonacci number using long method
        System.out.println("Long fib(10): " + fibOld(10));
    }
}

```

```
//print 92nd Fibonacci number using BigInteger method  
System.out.println("BigInt fib(92): " + fib(92));  
  
//print 92nd Fibonacci number using long method  
System.out.println("Long fib(92): " + fibOld(92));  
  
//print 93rd Fibonacci number using BigInteger method  
System.out.println("BigInt fib(93): " + fib(93));  
  
//print 93rd Fibonacci number using long method  
System.out.println("Long fib(93): " + fibOld(93));  
} //main  
} //class
```

The output from the above program will be:

```
BigInt fib(10): 55  
Long fib(10): 55  
BigInt fib(92): 7540113804746346429  
Long fib(92): 7540113804746346429  
BigInt fib(93): 12200160415121876738  
Long fib(93): -6246583658587674878
```

When you hit the 93rd Fibonacci number, you overflow the long method as the numbers get very large.

Again, like the factorial method, experiment with the numbers. It can be quite entertaining!

CHAPTER 12

Some Sorting Techniques

This chapter is the beginning of the data structures approach to Java programming. It covers Selection sort, Bubble sort, and Count sort.

TOPICS

- | | |
|--------------------------|-----|
| 1. <i>Selection Sort</i> | 279 |
| 2. <i>Bubble Sort</i> | 282 |
| 3. <i>Count Sort</i> | 284 |

When dealing with any kind of array, you may want to, in some way, sort the data they contain. You may sort them from lowest value to highest value (known as ascending), or from highest value to lowest value (known as descending).

There are many kinds of sorting techniques that can be used, but the three that will be discussed here are: **Selection Sort**, **Bubble Sort**, and **Count Sort**.

SELECTION SORT

The algorithm for Selection Sort is:

1. Search through each element.
2. Find the lowest or highest values.
3. Swap the elements.
4. REPEAT UNTIL FULLY SORTED.

Let's say that you have an array of integers set up as follows:

```
arr[] = {12, 9, 8, 3, 14};
```

And you wish to sort them in ascending order. Let's step through this one by one:

You want to start the search with the first element, which in this case is index 0. You have to then compare it with the remainder of the array, which in this case, begins with index 1. The current lowest value will be the value in the array at index 0.

```
arr[] = {12, 9, 8, 3, 14};      Lowest=12      Current=9  
Index_lowest=0
```

If the current value is less than the lowest value, store the index of the current value.

```
arr[] = {12, 9, 8, 3, 14};      Lowest=12      Current=9  
Index_lowest=1 (new lowest index since 9 < 12)
```

Repeat for the rest of the elements of the array:

```
arr[] = {12, 9, 8, 3, 14};      Lowest=9      Current=8  
Index_lowest=2
```

```
arr[] = {12, 9, 8, 3, 14};      Lowest=8      Current=3  
Index_lowest=3
```

```
arr[] = {12, 9, 8, 3, 14};      Lowest=3      Current=14
Index_lowest=3 (notice no change)
```

Now that you found the lowest value in the array, actually swap the elements. Recall that you are still dealing with index 0.

```
arr[] = {3, 9, 8, 12, 14};
```

Since this process began at index 0, move to index 1 and repeat the above steps.

```
arr[] = {3, 9, 8, 12, 14};      Lowest=9      Current=8
```

Index_lowest=2

```
arr[] = {3, 9, 8, 12, 14};      Lowest=8      Current=12
```

Index_lowest=2

```
arr[] = {3, 9, 8, 12, 14};      Lowest=8      Current=14
```

Index_lowest=2

In this case, there were no changes to the lowest index. Now swap the elements.

```
arr[] = {3, 8, 9, 12, 14};
```

There would be two more iterations of the above. You can accomplish this with loops:

```
for(int i = 0; i < length-1; i++){
    int indexMin = i;
    for(int j = i+1; j < length; j++){
        if(arr[j] < arr[indexMin]) indexMin = j;
    }
    //found the smallest elements index so swap here:
    if(arr[i] != arr[indexMin]){
        int temp = arr[i];
        arr[i] = arr[indexMin];
        arr[indexMin] = temp;
    }
}
```

EXAMPLE 1: *Selection Sort*

Below is an example program that demonstrates the use of Selection Sort via a method. It will sort an array of short integers in descending order (largest to smallest).

```

public class SelectionSort{
    //method for selection sort
    private static void selectionSort(short arr[]){
        int length = arr.length;
        short temp = 0;

        for(int i = 0; i < length-1; i++){
            int indexMax = i;
            for(int j = i+1; j < length; j++){
                if(arr[j] > arr[indexMax]) indexMax = j;
            }

            //found the smallest elements index so swap here:
            if(arr[i] != arr[indexMax]){
                temp = arr[i];
                arr[i] = arr[indexMax];
                arr[indexMax] = temp;
            }
        }
    } //method

    public static void main(String args[]){
        short nums[] = {5, 2, 3, 4, 7, 12, 9,
                       8, 9, 1, 9, 12, 16};

        //print the numbers before
        for(int i = 0; i < nums.length; i++)
            System.out.print(nums[i] + " ");
        System.out.println();
        System.out.println();

        //result should be ascending
        selectionSort(nums);

        //print the numbers after
        for(int i = 0; i < nums.length; i++)
            System.out.print(nums[i] + " ");
        System.out.println();
    } //main
} //class

```

Running the above program will produce the following output:

5 2 3 4 7 12 9 8 9 1 9 12 16

16 12 12 9 9 9 8 7 5 4 3 2 1

In the above, the array is sorted correctly, from largest to smallest.

BUBBLE SORT

This is the oldest sorting algorithm in computer science. A Bubble Sort will:

1. Compare one element with its neighbor.
2. Swap those elements if it is greater or less than.
3. REPEAT UNTIL SORTED.

Let's see a very small example first with this small array of integers. We want to sort them in ascending order.

```
arr[] = {9, 8, 2, 4};
```

Based on the algorithm, it first compares index 0 and index 1, or the values 9 and 8, respectively. Since 8 is less than 9, we want to swap them. Here is the revised array:

```
arr[] = {8, 9, 2, 4};
```

It now compares index 1 and index 2, or the values 9 and 2, respectively. Again, swap them.

```
arr[] = {8, 2, 9, 4};
```

It now compares index 2 and index 3, or the values 9 and 4, respectively. Again, swap them.

```
arr[] = {8, 2, 4, 9};
```

You have reached the end of the array. Simply start from the beginning. Here is the current array:

```
arr[] = {8, 2, 4, 9};
```

It compares index 0 and 1, or the values 8 and 2, respectively. Since 2 is less than 8, swap them:

```
arr[] = {2, 8, 4, 9};
```

It now compares index 1 and 2, or the values 8 and 4, respectively. Since 4 is less than 8, swap them:

```
arr[] = {2, 4, 8, 9};
```

It now compares index 2 and 3, or the values 8 and 9, respectively. Nothing happens, since 9 is not less than 8.

The above shows why you should not repeat all comparisons based on the length of the array. The first pass should be the entire array. The next should be one less. Then one less than that, etcetera and so forth...

Here are the loops that will perform the Bubble Sort:

```
for (int i = length-1; i >= 0; i--) {
    for (int j = 1; j <= i; j++) {
        if (numbers[j-1] > numbers[j]) {
            temp = numbers[j-1];
            numbers[j-1] = numbers[j];
            numbers[j] = temp;
        }
    }
}
```

The below program showcases the Bubble Sort for ascending order. Simply switch the less than operator with a greater than operator in the if statement above, should you wish to sort in descending order.

EXAMPLE 2: *Bubble Sort*

Below is an example program that demonstrates the use of Bubble Sort via a method. It will sort an array of double precision numbers in ascending order.

```
public class BubbleSort{
    //method to bubble sort an array
    private static void bubbleSort(double numbers[]){
        int length = numbers.length;
        double temp = 0;

        for (int i = length-1; i >= 0; i--) {
```

```

        for (int j = 1; j <= i; j++) {
            if (numbers[j-1] > numbers[j]) {
                temp = numbers[j-1];
                numbers[j-1] = numbers[j];
                numbers[j] = temp;
            }
        }
    }
} //method

public static void main(String args[]){
    double nums[] = {5.5, 4.2, 3.3, 4,
                     3.8, 9, 8.9, 1.02, 1.06};

    //print the numbers before
    for(int i = 0; i < nums.length; i++)
        System.out.print(nums[i] + " ");
    System.out.println();
    System.out.println();

    //result should be ascending
    bubbleSort(nums);

    //print the numbers after
    for(int i = 0; i < nums.length; i++)
        System.out.print(nums[i] + " ");
    System.out.println();
} //main
} //class

```

Running the above program produces the following output:

5.5 4.2 3.3 4.0 3.8 9.0 8.9 1.02 1.06

1.02 1.06 3.3 3.8 4.0 4.2 5.5 8.9 9.0

COUNT SORT

This is indeed an interesting sorting method, but it is the slowest if the original array features some large numbers. The algorithm for Count Sort is:

1. Find the largest number in the original array.
2. Declare a dynamic array of size $max+1$ and initialize all to 0, which will be used for the count of the numbers in the original array.

3. Loop through the original array and increment the appropriate index in the counter array.
4. Display the numbers from the original array based the total in the count array
OR sort the original array.

Let's see a small example of the above with this array:

```
arr[] = {2, 8, 4, 9, 2, 5};
```

It first searches for the max value in the array; in this case, that is the value of 9.

We now want to create a new array of size 10 (the max value plus one), and set that array to all zeroes.

```
arr[] = {2, 8, 4, 9, 2, 5};
arr2[] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
```

Next, we go through the array and increment the value in *arr2[]* at the index of the value in *arr[]*. So let's see the first step:

```
arr[] = {2, 8, 4, 9, 2, 5};
arr2[] = {0, 0, 1, 0, 0, 0, 0, 0, 0, 0};
```

At *arr2[2]*, we increased the value by 1. Let's see the next step:

```
arr[] = {2, 8, 4, 9, 2, 5};
arr2[] = {0, 0, 1, 0, 0, 0, 0, 0, 1, 0};
```

At *arr2[8]*, we increased the value by 1. The final result on that pass through will be:

```
arr[] = {2, 8, 4, 9, 2, 5};
arr2[] = {0, 0, 2, 0, 1, 1, 0, 0, 1, 1};
```

Which is correct, as there are two instances of the value 2 in *arr[]*, as well as one instance of the values 4, 5, 8, and 9. Now we would simply output two number 2s, 1 number 4, 5, 8, and 9 to the screen.

Here is a short program making use of the count sort:

```
public class CountSort{
    public static void main(String args[]) {
```

```

int arr[10] = {4,22,5,7,16,4,8,9,6,1};
int max = 0, i = 0;

//find the max number:
max = arr[0];
for(i = 1; i < 10; i++)
    if(arr[i] > max) max = arr[i];

//declare and initialize the array:
int sorted[] = new int[max+1];
for(i = 0; i <= max; i++)
    sorted[i] = 0;

//increment the counter based on the original array:
for(i = 0; i < 10; i++)
    sorted[ arr[i] ]++;

//display all the numbers in order:
for(i = 0; i <= max; i++){
    for(int j = 0; j < sorted[i]; j++)
        System.out.print(i + " ");
}

} //main
} //class

```

The output from running the above program will be:

1 4 4 5 6 7 8 9 16 22

You may also rewrite the last bit of the program that will certainly sort the original array. It may look something like this:

```

public class CountSort2{
    public static void main(String args[]){
        int arr[10] = {4,22,5,7,16,4,8,9,6,1};
        int max = 0, i = 0;

        //find the max number:
        max = arr[0];
        for(i = 1; i < 10; i++)
            if(arr[i] > max) max = arr[i];

        //declare and initialize the array:
        int sorted[] = new int[max+1];
        for(i = 0; i <= max; i++)
            sorted[i] = 0;
    }
}

```

```
for(i = 0; i < 10; i++)
    sorted[ arr[i] ] ++;

//new addition below here:
int index = 0;
for(i = 0; i <= max; i++){
    for(int j = 0; j < sorted[i]; j++)
        arr[index++] = i;
}

} //main
} //class
```

The only difference between this one and the previous version is the last part. We are now assigning the value back into the array after it has been counted.

CHAPTER 13

Working with Classes & Objects

This chapter gives an overview and understanding of the power of Object Oriented Programming. It covers more of the syntax of classes & objects, in addition to breaking down its parts (constructors, methods, variables, etc...)

TOPICS

1. <i>Rectangle Class</i>	289
2. <i>Fraction Class</i>	293
3. <i>Graph Class</i>	301
4. <i>Exercises</i>	306

Up until now, we have seen classes & objects entirely in different pieces, with different topics. Now, we have the understanding and capability to build an object related to a specific need.

RECTANGLE CLASS

For now, a simple class to consider making to get used to the premise of an object, is a class for a rectangle. By definition, a rectangle has 4 sides, where the pairs of opposite sides are equal, and all angles are right angles (90°). By convention, the dimensions of a rectangle are referred to as “length” and “width.”

Below is the start of the Rectangle class:

```
public class Rect{
    private double length, width;

    //default constructor setting length & width equal to 1.0
    public Rect(){
        length = width = 1.0;
    }

    //2 argument constructor
    public Rect(double a, double b){
        length = a;
        width = b;
    }
} //class
```

For now, the class contains two constructors, and two instance variables for the object. The default constructor will simply set the variables to the default value of 1.0 (since a rectangle cannot have any of the sides negative or 0). The two-argument constructor will set the variables to the values from the arguments.

As it is now, this is a complete class, but is absolutely pointless and does not perform any real function. Let's begin to expand on this class.

Get Methods/Accessors

Since the data variables are private, we need two methods here to return their values to us, so we can make use of them. These methods can be referred to as get methods (or accessor methods). We will add these two methods to the class above:

```
public double getLength() { return length; }
public double getWidth() { return width; }
```

Here, the methods are of type double, since each instance variable is of that same type. Both methods, when used, will simply return the value of the variable. The names of these methods should reflect the task you are trying to accomplish.

Set Methods/Mutators

Opposite of the get methods, we can use methods to change data inside an object. These methods are referred to as set methods (or mutator methods). Add these two methods after the get methods in the above class:

```
public void setLength(double len) {
    if(len <= 0.0)
        return;

    length = len;
}
public void setWidth(double w) {
    if(w <= 0.0)
        return;

    width = w;
}
```

Here, we need to do some error checking. As stated, any side of a rectangle needs to be greater than 0. The methods check their arguments' values, and if it is incorrect, we simply return out of the method, preventing incorrect data from being assigned to any instance variable.

Just like the accessor methods, the names of these set methods should reflect the task you are trying to accomplish.

Members

A **member** of the class Rect is simply any method that is part of the class. For now, let's add these two methods to the class, which deal with finding the area and perimeter of a Rect object.

In math, the area of a rectangle is found by returning the $length * width$. The perimeter is found by the formula $2(length + width)$, or in this case $2*length + 2*width$ after distributing.

```
public double Area(){
    return(length*width);
}

public double Perimeter(){
    return(2.0*length + 2.0*width);
}
```

These methods simply perform the appropriate operations for area and perimeter.

The Full Implementation

Below is the complete class piecing together the above snippets. It also provides a MainRect class for testing purposes.

```
//Rect.java
public class Rect{
    private double length, width;

    //default constructor setting length & width equal to 1.0
    public Rect(){
        length = width = 1.0;
    }

    //2 argument constructor
    public Rect(double a, double b){
        length = a;
        width = b;
    }

    //get methods:
    public double getLength(){ return length; }
```

```

        public double getWidth(){ return width; }

        //set methods:
        public void setLength(double len){
            if(len <= 0.0)
                return;

            length = len;
        }
        public void setWidth(double w){
            if(w <= 0.0)
                return;

            width = w;
        }

        //method to find the area of the rectangle
        public double Area(){
            return(length*width);
        }

        //method to find the perimeter of the rectangle
        public double Perimeter(){
            return(2.0*length + 2.0*width);
        }
    } //class

//MainRect.java
public class MainRect{
    public static void main(String args[]){
        Rect r1, r2;

        //define Rect objects
        r1 = new Rect();
        r2 = new Rect(5.0,3.0);

        //use get methods
        System.out.println(r1.getLength() + " " + r1.getWidth());
        System.out.println(r2.getLength() + " " + r2.getWidth());

        //find areas
        System.out.println("Area of r1 is: " + r1.Area());
        System.out.println("Area of r2 is: " + r2.Area());

        //make a change in r1
        r1.setLength(4.0);
        r1.setWidth(2.0);

        //display the perimeters of each
        System.out.println("Perimeter of r1 is: "
            + r1.Perimeter());
    }
}

```

```

        System.out.println("Perimeter of r2 is: "
                           + r2.Perimeter());
    } //main
} //class

```

The output from running the above MainRect class is:

```

1.0 1.0
5.0 3.0
Area of r1 is: 1.0
Area of r2 is: 15.0
Perimeter of r1: 12.0
Perimeter of r2: 16.0

```

FRACTION CLASS

Another great example of the concept of objects is to build a class that is responsible for holding information about a Fraction. By definition, a fraction, or rational number, is defined as k/l where k and l are integers, and l is not equal to zero, since the term would be undefined.

The beginning of the Fraction class is as follows:

```

public class Fraction{
    //instance variables for numerator and denominator
    private int n, d;

    //default constructor
    public Fraction(){
        n = 0;
        d = 1;
    }

    //one argument constructor which sets the numerator equal to i
    //and denominator equal to 1 (as the denominator cannot be 0)
    public Fraction(int i){
        n = i;
        d = 1;
    }

    //two argument constructor which sets n = i and d = j
    //it must check if j = 0 since the denominator must not be 0
    public Fraction(int i, int j){

```

```
    if(j == 0)
        return;
    n = i;
    d = j;
}
```

As mentioned, there are two parts to any fraction, the numerator and the denominator. They are simply declared as private integer variables *n* and *d*. These are private variables to prevent any mishaps of a user changing the data directly.

The first of the three constructors is the default constructor. This simply sets the numerator equal to 0 and the denominator equal to 1, since the denominator can never be equal to 0 by its rule.

The second of the three constructors is the one-argument constructor, for the case of a single integer. The argument represents the numerator of the fraction. The numerator and denominator are set the value of *i*.

The last of the three constructors represents the case of both the numerator and denominator. Here, we need some error checking. If the denominator is equal to 0, simply return from trying to create the object. Otherwise, set the values of the variables to the arguments of the constructor.

Get Methods

Here are the get methods for the Fraction class. They simply return the current value of the numerator and denominator.

```
public int getN(){ return n; }
public int getD(){ return d; }
```

Set Methods

Here are the set methods for the Fraction class. They simply set the current value of the numerator and denominator.

```
public void setN(int i){ n = i; }
public void setD(int i){
```

```
    if(i == 0)
        return;
    d = i;
}
```

The key here is the error checking in the setD() method. If the denominator is 0, it is an illegal value. Therefore, simply return out of the method to avoid an issue.

Arithmetic Members

A useful part of this class is to do arithmetic of two Fraction objects. The first in line is addition:

```
public Fraction add(Fraction f){
    Fraction t = new Fraction();
    t.n = n * f.d + f.n * d;
    t.d = d * f.d;
    return t;
}
```

Thinking about the math aspect of the addition of two fractions, let it be known that you need to find a common denominator between the two fractions in order for you to perform the addition properly. The above takes the shortcut of cross multiplying and placing that value as the numerator. The denominator is simply found by multiplying both denominators. These values are placed into the temporary Fraction to be returned to represent the addition of the fractions.

The next operation is subtraction:

```
public Fraction subtract(Fraction f) {
    Fraction t = new Fraction();
    t.n = n * f.d - f.n * d;
    t.d = d * f.d;
    return t;
}
```

This is the additive inverse of the addition method. It simply finds the common denominator and subtracts the values instead of adding them. These values are placed

into the temporary Fraction to be returned, which represents the subtraction of the two fractions.

The next operation is multiplication:

```
public Fraction multiply(Fraction f){  
    Fraction t = new Fraction();  
    t.n = n * f.n;  
    t.d = d * f.d;  
    return t;  
}
```

By the rules of multiplying fractions, you simply multiply across for both the numerator and the denominator.

Finally, the last operation is division:

```
public Fraction divide(Fraction f){  
    Fraction t = new Fraction();  
    t.n = n * f.d;  
    t.d = d * f.n;  
    return t;  
}
```

For division of two Fractions, you need to multiply the fraction by the reciprocal of the other one.

Method Overloading

Here are two methods that you should add to the above class:

```
public String toString(){  
    return n + " / " + d;  
}  
public boolean equals(Fraction f){  
    return n * f.d == d * f.n;  
}
```

The first of the above is the overloaded `toString()` method. It simply returns a “fraction” string, placing the numerator over the denominator.

The second is an `equals()` method, which is used to check if two objects are exact equals (such as $\frac{2}{3}$ and $\frac{2}{3}$), or if their values are equal (such as $\frac{1}{2}$ and $\frac{3}{6}$).

Reducing Fractions

Now that the arithmetic methods are defined, we should also define a method that will allow us to reduce a fraction. This means we also need to define another method that will find the **greatest common factor** (or **greatest common divisor**) between two numbers.

```
private int gcf(int a, int b){  
    if(b == 0)  
        return Math.abs(a);  
    else  
        return gcf(b, a%b);  
}
```

This method follows Euclid’s algorithm. Let’s do a proof using the two numbers 54 and 24:

Proof recursively:

```
gcf(54, 24)  
  gcf(24, 54 % 24) → also gcf(24, 6)  
    gcf(6, 0) → base case  
  6  
6
```

The method above returns 6, which, in turn, is the greatest common factor among 54 and 24. This method will be utilized when we want to reduce a fraction.

```
private void reduce(){  
    //the case where it is 0  
    if(n == 0){  
        n = 0;
```

```

        d = 1;
    }

    int c = gcf(Math.abs(n), Math.abs(d));
    n /= c;
    d /= c;
}

```

We need to check if the numerator is 0. If true, make sure the fraction is set to 0/1, since the denominator cannot be 0. We then find the gcf() of the two values n and d . Once found, you now need to divide the numerator and denominator by that value. The end result is a reduced fraction.

The reduce method will be utilized in each of the arithmetic methods we defined. The full implementation below contains the modification of each method.

The Full Implementation

Below is the complete class for a Fraction. A main class has been provided as a way of testing this class.

```

//Fraction.java
public class Fraction{
    //instance variables for numerator and denominator
    private int n, d;

    //default constructor
    public Fraction(){
        n = 0;
        d = 1;
    }

    //one argument constructor which sets the numerator equal to i
    //and denominator equal to 1 (as the denominator cannot be 0)
    public Fraction(int i){
        n = i;
        d = 1;
    }

    //two argument constructor which sets n = i and d = j
    //it must check if the j = 0 since the denominator must not be 0
    public Fraction(int i, int j){
        if(j == 0)
            return;
        else
            reduce();
    }

    //reduces the fraction to its lowest terms
    private void reduce(){
        int c = gcf(Math.abs(n), Math.abs(d));
        n /= c;
        d /= c;
    }

    //returns the numerator
    public int getNumerator(){
        return n;
    }

    //returns the denominator
    public int getDenominator(){
        return d;
    }

    //sets the numerator to i
    public void setNumerator(int i){
        n = i;
    }

    //sets the denominator to j
    public void setDenominator(int j){
        d = j;
    }

    //sets both the numerator and denominator to i
    public void setFraction(int i){
        n = i;
        d = 1;
    }

    //sets both the numerator and denominator to i
    public void setFraction(int i, int j){
        n = i;
        d = j;
    }

    //returns a string representation of the fraction
    public String toString(){
        return n + "/" + d;
    }
}

```

```

        n = i;
        d = j;
    }

    //get/set methods
    public int getN(){ return n; }
    public int getD(){ return d; }

    public void setN(int i){ n = i; }
    public void setD(int i){
        if(i == 0) return;

        d = i;
    }

    //method to add, then reduce a fraction
    public Fraction add(Fraction f){
        Fraction t = new Fraction();
        t.n = n * f.d + f.n * d;
        t.d = d * f.d;
        t.reduce();
        return t;
    }

    //method to subtract, then reduce a fraction
    public Fraction subtract(Fraction f){
        Fraction t = new Fraction();
        t.n = n * f.d - f.n * d;
        t.d = d * f.d;
        t.reduce();
        return t;
    }

    //method to multiply, then reduce a fraction
    public Fraction multiply(Fraction f){
        Fraction t = new Fraction();
        t.n = n * f.n;
        t.d = d * f.d;
        t.reduce();
        return t;
    }

    //method to divide, then reduce a fraction
    public Fraction divide(Fraction f){
        Fraction t = new Fraction();
        t.n = f.n * d;
        t.d = n * f.d;
        t.reduce();
        return t;
    }
}

```

```

public String toString() {
    return n + " / " + d;
}

public boolean equals(Fraction f) {
    return n * f.d == d * f.n;
}

private int gcf(int a, int b) {
    if(b == 0)
        return Math.abs(a);
    else
        return gcf(b, a%b);
}

private void reduce() {
    //the case where it is 0
    if(n == 0) {
        n = 0;
        d = 1;
    }

    int c = gcf(Math.abs(n), Math.abs(d));
    n /= c;
    d /= c;
}
} //class

//MainFraction.java
public class MainFraction{
    public static void main(String args[]){
        Fraction f1, f2;

        f1 = new Fraction();
        f2 = new Fraction(4,5);

        System.out.println(f1);
        System.out.println(f2);

        f1.setN(9);
        f1.setD(4);

        System.out.println("f1 + f2 = " + f1.add(f2));
        System.out.println("f1 - f2 = " + f1.subtract(f2));
        System.out.println("f1 * f2 = " + f1.multiply(f2));
        System.out.println("f1 / f2 = " + f1.divide(f2));

        if(f1.equals(f2)){
            System.out.println("They are equals!");
        }else{
            System.out.println("Not equals!");
    }
}

```

```
        }
    } //main
} //class
```

The output from running the above MainFraction will be:

```
0 / 1
4 / 5
f1 + f2 = 61 / 20
f1 - f2 = 29 / 20
f1 * f2 = 9 / 5
f1 / f2 = 16 / 45
Not equals!
```

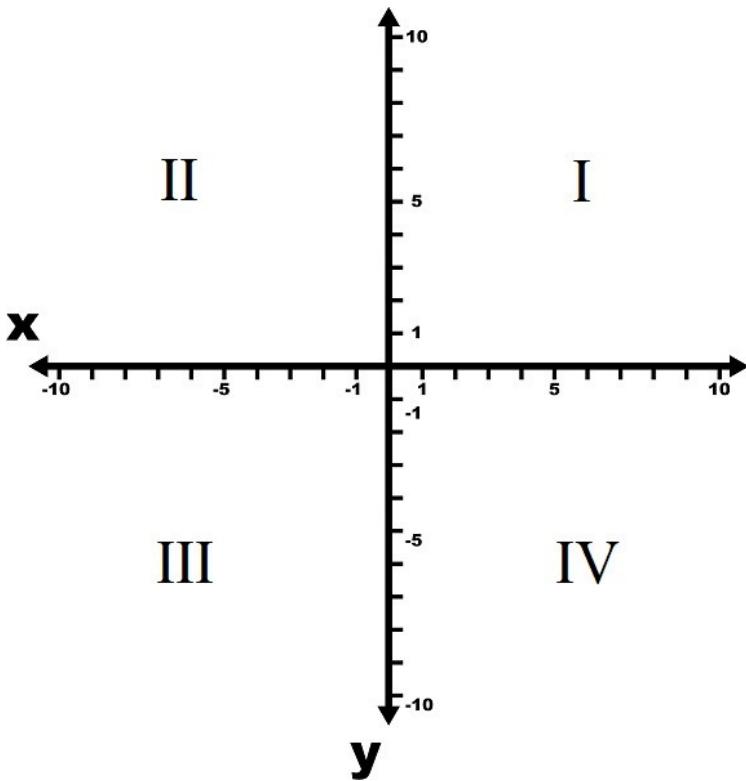
Experiment with the above fractions to see various outputs.

GRAPH CLASS

In mathematics, a coordinate plane, or graph, is made up of an *x-axis* and *y-axis*. Any point on that plane is (x,y). Points on the plane can be positive or negative, as well as decimals. Using this information, we want to write a class called **Graph** that will feature the following:

- a) A default constructor, setting both *x* and *y* equal to 0.
- b) A two-argument constructor, setting *x* and *y* to the arguments. No error checking is needed, since a point can be positive, negative or zero.
- c) Two set methods for each of the data variables.
- d) Two get methods for each of the data variables.
- e) A method called *distance()*, which will calculate the distance between two points. The distance formula is as follows:
$$\sqrt{(x_2-x_1)^2 + (y_2-y_1)^2}$$
- f) A method called *midpoint()*, which will take another Graph object as its argument, and calculate the midpoint of those two points. NOTE: The method is of type Graph. The midpoint formula is:
$$((x_1+x_2)/2, (y_1+y_2)/2)$$
- g) A method called *quadrant()*, which will print a message stating which quadrant on the plane the coordinate falls (see grid below to determine quadrants).

Here is what a coordinate plane looks like:



Where in the above, the Roman numerals are the quadrants of the plane.

Let's dive right in to it! The Graph class will begin as such:

```
public class Graph{
    //instance variables for point x and point y
    private double x, y;

    //default constructor
    public Graph(){
        x = y = 0.0;
    }

    //argument constructor, indicating points x and y
    public Graph(double a, double b){
        x = a;
        y = b;
    }
}
```

This satisfies points *a* and *b* above. Now let's get to points *c* and *d*, for the get and set methods:

```
//get/set methods
public void setX(double a){ x = a; }
public void setY(double b){ y = b; }

public double getX(){ return x; }
public double getY(){ return y; }
```

Point *e* above is asking us to calculate the distance between two points. Recall the distance formula:

$$\sqrt{(x_2-x_1)^2 + (y_2-y_1)^2}$$

Fortunately, we have the built-in Math library to come to our rescue for square root!

```
//method to find the distance between two points on a Graph
public double distance(Graph g) {
    double tx = (g.x-x) * (g.x-x);
    double ty = (g.y-y) * (g.y-y);
    return Math.sqrt(tx + ty);
}
```

Point *f* above is asking us to calculate the midpoint between two points. Recall the midpoint formula:

$$((x_1+x_2)/2, (y_1+y_2)/2)$$

```
//method to find the midpoint between two points on a Graph
public Graph midpoint(Graph g) {
    Graph temp = new Graph();
    double tx = (g.x + x) / 2.0;
    double ty = (g.y + y) / 2.0;
    temp.setX(tx);
    temp.setY(ty);
    return temp;
}
```

The final point is asking us to write a method called *quadrant()*, that will determine on which of the four quadrants a particular point lies (see graph in beginning of the section for reference). This method will use a series of if/else statements to determine that. If the x,y coordinates are 0,0, you are at what's called the origin of the graph.

```
//method to find the midpoint between two points on a Graph
public void quadrant(){
    if(x > 0 && y > 0)
        System.out.println("Lies in Quadrant 1");
    else if(x < 0 && y > 0)
        System.out.println("Lies in Quadrant 2");
    else if(x < 0 && y < 0)
        System.out.println("Lies in Quadrant 3");
    else if(x > 0 && y < 0)
        System.out.println("Lies in Quadrant 4");
    else if(x == 0 && y > 0)
        System.out.println("Lies on positive y-axis");
        else if(x == 0 && y < 0)
        System.out.println("Lies on negative y-axis");
    else if(x < 0 && y == 0)
        System.out.println("Lies on negative x-axis");
    else if(x > 0 && y == 0)
        System.out.println("Lies on positive x-axis");
    else
        System.out.println("Lies at origin 0,0");
}
```

The Full Implementation

Here is the complete Graph class, with a class provided for testing purposes.

```
//Graph.java
public class Graph{
    //instance variables for point x and point y
    private double x, y;

    //default constructor. Sets points to origin
    public Graph(){
        x = y = 0.0;
    }
    //argument constructor, indicating points x and y
    public Graph(double a, double b){
        x = a;
        y = b;
    }
}
```

```

//get/set methods
public void setX(double a){ x = a; }
public void setY(double b){ y = b; }

public double getX(){ return x; }
public double getY(){ return y; }

//method to find the distance between two points on a Graph
public double distance(Graph g){
    double tx = (g.x-x) * (g.x-x);
    double ty = (g.y-y) * (g.y-y);
    return Math.sqrt(tx + ty);
}

//method to find the midpoint between two points on a Graph
public Graph midpoint(Graph g){
    Graph temp = new Graph();
    double tx = (g.x + x) / 2.0;
    double ty = (g.y + y) / 2.0;
    temp.setX(tx);
    temp.setY(ty);
    return temp;
}

//method to find the midpoint between two points on a Graph
public void quadrant(){
    if(x > 0 && y > 0)
        System.out.println("Lies in Quadrant 1");
    else if(x < 0 && y > 0)
        System.out.println("Lies in Quadrant 2");
    else if(x < 0 && y < 0)
        System.out.println("Lies in Quadrant 3");
    else if(x > 0 && y < 0)
        System.out.println("Lies in Quadrant 4");
    else if(x == 0 && y > 0)
        System.out.println("Lies on positive y-axis");
    else if(x == 0 && y < 0)
        System.out.println("Lies on negative y-axis");
    else if(x < 0 && y == 0)
        System.out.println("Lies on negative x-axis");
    else if(x > 0 && y == 0)
        System.out.println("Lies on positive x-axis");
    else
        System.out.println("Lies at origin 0,0");
}
//overloaded toString() method
public String toString() {
    return "Currently at (" + x + ", " + y + ")";
}
} //Graph

```

```
//MainGraph.java
public class MainGraph{
    public static void main(String args[]) {
        Graph g1, g2;
        g1 = new Graph();
        g2 = new Graph(5, -2);

        System.out.println(g1);
        System.out.println(g2);

        g1.setX(-8);
        g2.setY(0);
        g1.quadrant();
        g2.quadrant();

        System.out.println(g1.midpoint(g2));
        System.out.println(g2.distance(g1));
    }
} //class
```

The output from running the above MainGraph is:

```
Currently at (0.0, 0.0)
Currently at (5.0, -2.0)
Lies on negative x-axis
Lies on positive x-axis
Currently at (-1.5, 0.0)
13.0
```

As you develop more and more objects, the overall concept of objects will become much clearer and easier to work with. The three examples here in this chapter were just some of them to get you started on understanding classes and objects.

EXERCISES

DIRECTIONS: Answer each question below to the best of your ability. Solutions on page 650.

Problem 1:

By the mathematical property, a square is a figure with all 4 sides equal to each other. Write a class called **Square** that will include the following:

- A default constructor, setting the side to 1.

- b) A one-argument constructor, setting the side equal to the argument. Some error checking is needed here to make sure the side is not less than or equal to 0.
- c) A set method called *setSide()* that will set the side of the square manually. Again, do the same error checking as part b.
- d) An accessor called *getSide()* that will return the side of the square.
- e) A method called *area()* that returns a value representing the area of the square (*side * side*).
- f) A method called *perimeter()* that returns a value representing the perimeter of the square (*4 * side*).

Problem 2:

Write a class called ***Book*** that will be used to hold information about books. The class will include variables for the title, author, price and year released. The class will also include the following:

- a) A default constructor setting all the class variables to default values.
- b) A 4-argument constructor setting the class variables to the argument values.
- c) Get and set methods for the variables.
- d) A member method called *compare()* that will compare the price of two books. The argument of the method will be another Book object. It will return -1 if the current object's price is smaller than the arguments price; 0 if they are equal; and 1 otherwise.
- e) An overloaded *toString()* method that will output the contents of the book in a nice format.

Additional Problems

DIRECTIONS: Answer each problem below to the best of your ability. Solutions are not provided in this book.

Problem 3:

Write a class called ***Weight*** that will keep track of the weight of an object. The class will have variables for the pounds and ounces (of type int). The class will also include the following:

- a) A default constructor that will set the pounds and ounces equal to 0.
- b) A two-argument constructor that will set the pounds and ounces equal to the arguments of the constructor.
- c) Get and set methods for the pounds and ounces.

- d) A member method called *adjust()* that will adjust the pounds and ounces if the user enters a value of more than 15 for ounces. This method will be called in the set method for the ounces and the two argument constructor.
- e) A member method called *compare()* that will compare two Weight objects. The parameter of the method will be another Weight object. It will return -1 if the current object's weight is smaller than the arguments weight; 0 if they are equal; and 1 otherwise.
- f) An overloaded *toString()* method that will output the values in a nice format.

Problem 4:

Write a class called *Math* that will include the following:

- a) An instance variable called *num* that is of type long.
- b) A default constructor that will set the value of the variable to 0.
- c) A one-argument constructor that will set the value of the variable equal to the argument of the constructor. If the argument is less than or equal to 0, an *IllegalArgumentException* should be thrown.
- d) A get method for the variable.
- e) A method called *factorial()* that will return the factorial of the instance variable.
- f) A method called *power()* that will return the variable to that given power.
- g) A method called *half()* that will return a double precision value of half of that variable.

CHAPTER 14

Inheritance

This chapter will cover a basic understanding of what it means for a class to inherit properties from another, on both a single level and a multi-level.

TOPICS

1. <i>Introduction</i>	310
2. <i>Super Keyword</i>	310
3. <i>This Keyword</i>	311
4. <i>GetClass() & GetName()</i>	311
5. <i>Inheritance Examples</i>	312
6. <i>Exercises</i>	331

In Java, a class can **inherit all public** methods and instance variables from another class by extending it, with the use of the keyword **extends**. Most generally, here is how to extend a class from another:

```
public class Class_sub extends Class_super{  
    //code here  
}
```

Where in the above, ***Class_sub*** is an appropriate name for the **subclass** (or **child class**); and ***Class_super*** is an appropriate name for the **super** class (or **parent class**).

To help understand the concept a little more, let's see some real world examples:

```
public class Apple extends Fruit{  
    //code here  
}  
  
public class Wheel extends Car{  
    //code here  
}  
  
public class Yankees extends Baseball{  
    //code here  
}  
  
public class Cat extends Animal{  
    //code here  
}  
  
Etc...
```

In human terms, an apple is a type of fruit; a wheel is part of a car; the Yankees are a baseball team; and a cat is an animal. All are pieces of a larger entity.

SUPER KEYWORD

The keyword **super** will be used when referring to the super class (or parent class) of an object. This can reference the constructor when used like this:

```
super( argument(s) );
```

Or if accessing another method:

```
super.methodName( argument(s) );
```

Or if accessing a public instance variable:

```
super.variableName = value;
```

What is **NOT** inherited directly from the super class are the constructors, since they are strictly used by that class (the parent class).

THIS KEYWORD

The keyword **this** will reference the current instance of an object where it appears. It will allow you to use the classes' methods if used like this:

```
this.methodName();
```

Or access the current objects instance variable:

```
this.variableName = value;
```

GETCLASS() & GETNAME() METHOD

When referring to classes, there is a method called `getClass()` that will return the **current** class it is called in.

Inside of the getClass() method is another method called getName(). This will return a String representing the name of that class. For example, if the name of your subclass was Dog, the String "Dog" would be returned.

INHERITANCE EXAMPLES

The best way to learn inheritance is to practice it. Pay attention to each example closely.

EXAMPLE 1: *Animals*

```
//Animal.java
public class Animal{
    //instance variables
    private String name;
    private int age;

    //two argument constructor for name and age
    public Animal(String n, int a){
        name = n;
        age = a;
    }

    //get methods
    public String getName() { return name; }
    public int getAge() { return age; }
} //Animal

//Dog.java
public class Dog extends Animal{
    //two argument constructor for name and age
    public Dog(String n, int a){
        super(n, a);
    }
    //a dog goes woof!
    private String sound() { return "Woof!"; }

    //overloaded toString method
    public String toString() { return "My name is "
        + super.getName() + " I am a "
        + this.getClass().getName() + " and I " + sound(); }
} //Dog

//Cat.java
public class Cat extends Animal{
    //two argument constructor for name and age
```

```

public Cat(String n, int a){
    super(n, a);
}
//a cat meows!
private String sound() { return "Meow!"; }

//overloaded toString method
public String toString() { return "My name is "
    + super.getName() + " I am a "
    + this.getClass().getName() + " and I " + sound(); }
}

} //Cat

//Example1.java
public class Example1{
    public static void main(String args[]){
        Dog d = new Dog("Ralph", 5);
        Cat c = new Cat("Fluffy", 10);

        //print current d and c objects
        System.out.println(d);
        System.out.println(c);

        //make a new instance of each
        d = new Dog("Butch", 4);
        c = new Cat("Max", 3);

        //print current d and c objects
        System.out.println(d);
        System.out.println(c);
    } //main
} //class

```

The output from running the above program is:

```

My name is Ralph I am a Dog and I Woof!
My name is Fluffy I am a Cat and I Meow!
My name is Butch I am a Dog and I Woof!
My name is Max I am a Cat and I Meow!

```

Let's explore why. Firstly, with any type of inheritance, it helps to identify which class is the super class, and which classes are the subclasses. In this example, the *Animal* class is the super class, whereas Dog and Cat are the subclasses. It also helps to identify which methods or variables the subclasses will inherit from the parent class. The charts below will clarify this:

ANIMAL	
<u>Contains:</u> getName() getAge()	<u>Inherits:</u> NONE (Parent class)

DOG	
<u>Contains/Local:</u> sound() toString()	<u>Inherits (from Animal)</u> getName() getAge()

CAT	
<u>Contains/Local:</u> sound() toString()	<u>Inherits (from Animal):</u> getName() getAge()

The constructor of each of subclass makes a call, via the super keyword, to the parent classes' constructors, passing the respective values. Additionally, each subclass has an overloaded `toString()` method, which will print information about the current instance of the respective object.

Running the `Example1` class will produce the output above.

EXAMPLE 2: *Mystery Math*

Read through each class below. Take careful note of “hidden” set methods!

```
//MathOps.java
public class MathOps{
    //instance variables for the two numbers
    private double n1, n2;

    //two argument constructor, taking the respective values
    //for the two numbers
    MathOps(double n1, double n2) {
        this.n1 = n1;
        this.n2 = n2;
    }
}
```

```

//set methods
public void setN1(double n1) { this.n1 = n1; }
public void setN2(double n2) { this.n2 = n2; }

//member methods
private double sum() { return n1+n2; }
private double product() { return n1*n2; }
private double diff() { return n1-n2; }
private double div() { return n1/n2; }

//I wonder what happens
public double mystery() {
    return sum() + product();
}

//The mystery deepens
public double mystery2() {
    return div() - diff();
}
} //MathOps

//MathClass.java
public class MathClass extends MathOps{
    //two argument constructor, taking in the values for n1 and n2
    public MathClass(double n1, double n2){
        super(n1, n2);
        setN1(n1-3);
    }

    //mystery method
    public void mystery3() {
        System.out.println("It's a mystery!");
        setN2(6);
        System.out.println(mystery());
    }

    //another mystery method
    public void mystery4() {
        System.out.println("It's a bigger mystery!");
        setN1(12);
        System.out.println(mystery2());
    }
}

//Example2.java
public class Example2{
    public static void main(String args[]){
        MathClass m = new MathClass(6, 3);

        System.out.println(m.mystery());
    }
}

```

```

        System.out.println(m.mystery2());
        m.mystery3();
        m.mystery4();
    } //main
} //class

```

Let's create a chart to determine what each class contains and inherits.

MathOps	
<u>Contains/Local:</u>	<u>Inherits:</u>
setN1() setN2() sum() diff() product() div() mystery() mystery2()	NONE (Parent class)

MathClass	
<u>Contains/Local:</u>	<u>Inherits from MathOps:</u>
mystery3() mystery4()	setN1() setN2() mystery() mystery2()

The output from running the Example2 class is:

```

15.0
1.0
It's a mystery!
27.0
It's a bigger mystery!
-4.0

```

As always, let's start in the main() method. We create a new *MathClass* object with initial values of 6 and 3 (*n1* and *n2*, respectively). However, in the *MathClass*

constructor, we set the value of *n1*, via the *setN1()* method, to the argument value, minus 3 (here, 6-3 or 3). Even though we gave it the initial value of 6, we immediately set it to a different value.

Proceeding down the *main()* method, we print out the value returned from the inherited *mystery()* method from the MathOps class. That method returns the sum, plus the product of the two numbers. Recall, *n1* and *n2* are both set at 3 at the moment. The method returns 15.0, $((3.0 + 3.0) + (3.0 * 3.0))$, and the value is printed to the console.

Next, we call the *mystery2()* method that we inherited from the MathOps class. That method returns the quotient, and the difference of the two variables. Both *n1* and *n2* are still set to 3 at this point. The method returns 1.0, $((3.0/3.0) - (3.0-3.0))$, and the value is printed to the console.

Next, we invoke the local *mystery3()* method contained in the MathClass object. This is not an inherited method from the MathOps class, as it is local to MathClass. Here, the method is void, so we cannot place it inside a *System.out* statement. It first prints the message “It’s a mystery!” to the console. After that, we set the value of *n2* to 6, via the inherited *setN2()* method. Finally, we invoke the inherited *mystery()* method, now dealing with the value of 3 and 6 for *n1* and *n2*, respectively. That method returns 27.0, $((3.0 + 6.0) + (3.0 * 6.0))$, and the value is printed to the console.

Finally, we call the local *mystery4()* method contained in the MathClass object. Again, this is not an inherited method from the MathOps class, as it is local to MathClass. Here, the method first prints “It’s a bigger mystery!” to the console. After that, we set the value of *n1* to 12, via the inherited *setN1()* method. Finally, we invoke the inherited *mystery2()* method, now dealing with 12 and 6 for the values of *n1* and *n2*, respectively. That method returns -4.0, $((12.0/6.0) - (6.0-12.0))$, and prints that value to the console.

EXAMPLE 3: Sports Teams

There are many different kinds of sports in existence: basketball, football, bowling, and tennis, just to name a few. Within every sport there are teams.

We want to write a program that will contain the following, and perform the needed tasks:

1. Build a parent class called **Team** that will:

- a. Have an instance variable called *name*, which holds the name of the team; *num_wins*, which keeps track of the wins a team has; and *num_losses*, keeping track of the losses a team has.
 - b. The constructor will be one-argument, taking the team name and setting the *num_wins* and *num_losses* variables to 0.
 - c. Have get methods for each of the instance variables.
 - d. Have set methods for *num_wins* and *num_losses*.
2. Build a class called **Basketball**, which will be a subclass of **Team**, that will:
 - a. Have an instance variable called *coach*, to hold the name of the coach of the team; a boolean variable called *season_played*, initially set to false in the constructor; and a final double variable called *NUM_GAMES* set to 82, which represents the number of games in an NBA basketball season.
 - b. The constructor will be two-arguments, taking the name of the team and the name of the coach, and setting the *season_played* variable to false.
 - c. A private method called *winPercent()* that will determine the winning percentage of the team (this will use the get method for the *num_wins* variable above).
 - d. A method called *playSeason()* that will simulate a season of *NUM_GAMES*. This method will need the Random library, and its method *nextDouble()*. If the generated number is strictly less than 0.5, increase the number of wins, else increase the number of losses, and set the appropriate values, via the parent classes' set methods.
 - e. An overloaded *toString()* method, printing appropriate information. If a season has been played, print the number of wins, losses, and the winning percentage. If the winning percentage is greater than or equal to 0.5, print "Winning season!" as part of the String, else print "Losing Season!"
3. Build a class called **Baseball**, which will be a subclass of **Team**, that will:
 - a. Contain the same instance variables as the Basketball class above, with the *NUM_GAMES* variable being set to 162, which represents the number of games in an MLB season.
 - b. Contain the same methods as the Basketball class (*winPercent()*, *playSeason()* and *toString()*).

Whew! Seems like a lot right? Well, it's not! Let's get to work!

Team Class

Following the instructions from item 1 above, the constructor of the class will take the name of the team. The wins and losses will be set to zero initially, as the program will simulate a season for the respective team. We also create the get/set methods needed.

```
//Team.java
public class Team{
    //instance variables. Every team in a sport
    //has a number of wins, losses and a team name

    private int num_wins, num_losses;
    private String name;

    //one argument constructor taking the team name
    //sets the num_wins and num_losses to 0 as no season has been
    //played yet.
    public Team(String n) {
        name = n;
        num_wins = num_losses = 0;
    }

    //get methods
    public String getName() { return name; }
    public int getWins() { return num_wins; }
    public int getLosses() { return num_losses; }

    //set methods
    public void setWins(int w) { num_wins = w; }
    public void setLosses(int l) { num_losses = l; }
} //Team
```

Now we need to build each subclass.

Basketball Class

Per item 2 in the instructions, we want the *Basketball* class to be a subclass of *Team*. That is accomplished with the `extends` keyword. We also want the instance variables for *coach*, *NUM_GAMES* and *season_played*.

The constructor of the class will be two arguments, taking the name of the team and the name of the coach. We must invoke the super classes' constructors before we do anything else (per the rules of inheritance). The super classes' constructors take one argument, which in that case, is the name of the team.

The class will begin like the below:

```
import java.util.Random;
public class Basketball extends Team{
    //each team has a coach
    private String coach;

    //NBA rules state there are 82 games in a season
    private final double NUM_GAMES = 82;

    //boolean to check if we played the season yet
    private boolean season_played;

    //two argument constructor, taking the team name & coach
    Basketball(String n, String c){
        super(n);
        coach = c;
        season_played = false;
    }
}
```

To find a team's winning percentage, you divide the number of wins the team has by the number of games played. It will give you a decimal percentage, so we know we want to use a double precision value. Say that the team played 10 games and won 6 of them. The winning percentage is $.600$ ($6/10 = 0.600$).

The method is defined below:

```
//method to calculate teams' winning percentage
private double winPercent() {
    return (double)getWins() / NUM_GAMES;
}
```

We cast to a double the result of the *getWins()* method, since the *num_wins* variable in the *Team* class is an int value.

Continuing on down, we want to create the *playSeason()* method. Here, we invoke the Random object we need to generate the random result of a game. If the generated decimal value is less than 0.5, we won a game, otherwise we lost. We also need a for loop, running from 1 to *NUM_GAMES*. Within each iteration of the loop, we generate the random value we need, and increase the wins and losses accordingly.

After the loop executes, we want to set the recorded number of wins and losses via the set methods we created in the Team class (*setWins()* and *setLosses()*). The method is defined below:

```
//method to simulate the baseball teams' season
public void playSeason() {
    Random r = new Random();
    int wins = 0, losses = 0;

    //loop through the number of games in the season
    for(int i = 1; i <= NUM_GAMES; i++) {
        //50/50 chance of winning
        if(r.nextDouble() <= 0.5) {
            wins++;
        } else {
            losses++;
        }
    }
    //set wins and losses for the team
    this.setWins(wins);
    this.setLosses(losses);

    //season has been played so set variable to true
    season_played = true;
}
```

Next, we want to create the overloaded *toString()* method for the class. If the season has been played, we will print the number of wins, losses, and the teams' winning percentages. Otherwise, we print basic information.

The method is defined below:

```
//overloaded toString method. Want to check if season has been
//played yet. If so, print the wins, losses and win percentage
public String toString() {
    //get winning percentage for team
    double wp = winPercent();

    //response String
    String ans = "";

    //if we played the season, then print appropriate info:
    if(season_played) {
```

```

        ans += this.getClass().getName() + " team: "
            + getName() + "\nCoached by: " + coach + "\n"
            + "Wins: " + getWins() + "\n"
            + "Losses: " + getLosses() + "\n"
            + "Win percent: " + wp;

        //determine if we had a winning or losing season
        //a winning season is a win percentage of 0.5 or
        //greater. otherwise it's a losing season
        if(wp < 0.5)
            ans += "\nLosing season!";
        else
            ans += "\nWinning season!";

    }else {

        //we did not play season so print basic information
        ans += this.getClass().getName() + " team: "
            + getName() + "\nCoached by: " + coach + "\n"
            + "Awaiting season results";
    }

    //return final String
    return ans + "\n";
}

}//Basketball class

```

Baseball Class

The Baseball class will contain the same overall structure as the Basketball class. Here is the complete implementation:

```

import java.util.Random;
public class Baseball extends Team{
    //each team has a coach
    private String coach;

    //MLB rules state there are 162 games in a season
    private final double NUM_GAMES = 162;

    //boolean to check if we played the season yet
    private boolean season_played;

    //two argument constructor, taking the team name & coach
    public Baseball(String n, String c){
        super(n);
    }
}

```

```

        coach = c;
        season_played = false;
    }

    //method to calculate teams' winning percentage
    private double winPercent() {
        return (double)getWins() / NUM_GAMES;
    }

    //method to simulate the baseball teams' season
    public void playSeason() {

        Random r = new Random();
        int wins = 0, losses = 0;

        //loop through the number of games in the season
        for(int i = 1; i <= NUM_GAMES; i++) {
            //50/50 chance of winning
            if(r.nextDouble() <= 0.5) {
                wins++;
            } else {
                losses++;
            }
        }
        //set wins and losses for the team
        this.setWins(wins);
        this.setLosses(losses);

        //season has been played so set variable to true
        season_played = true;
    }

    //overloaded toString method. Want to check if season has been
    //played yet. If so, print the wins, losses and win percentage
    public String toString() {

        //get winning percentage for team
        double wp = winPercent();

        //response String
        String ans = "";

        //if we played the season, then print appropriate info:
        if(season_played) {

            ans += this.getClass().getName() + " team: "
                + getName() + "\nCoached by: " + coach + "\n"
                + "Wins: " + getWins() + "\n"
                + "Losses: " + getLosses() + "\n"
                + "Win percent: " + wp;
        }
    }
}

```

```

        //determine if we had a winning or losing season
        //a winning season is a win percentage of 0.5 or
        //greater. otherwise it's a losing season
        if(wp < 0.5)
            ans += "\nLosing season!";
        else
            ans += "\nWinning season!";

    } else {

        //we did not play season so print basic information
        ans += this.getClass().getName() + " team: "
            + getName() + "\nCoached by: " + coach + "\n"
            + "Awaiting season results";
    }
    //return final String
    return ans + "\n";
}
} //Baseball class

```

Testing Class

Let's now build a separate PlayBall class that will create some output.

```

public class PlayBall{
    public static void main(String args[]){
        //setup new teams
        Basketball team1 =
            new Basketball("Knicks", "David Fizdale");
        Baseball team2 =
            new Baseball("Mets", "Terry Collins");
        Team team3 =
            new Basketball("Heat", "Eric Spolstra");

        //print current teams
        System.out.println(team1);
        System.out.println(team2);
        System.out.println(team3);

        //play the season
        team1.playSeason();
        team2.playSeason();

        //print current teams
        System.out.println(team1);
        System.out.println(team2);
        System.out.println(team3);
    }
}

```

```

    } //main
} //class

```

As shown in examples 1 & 2, let's determine which methods are contained in each class:

Team	
<u>Contains/Local:</u>	<u>Inherits:</u>
<u>Contains/Local:</u> getName() getWins() getLosses() setWins() setLosses()	<u>Inherits:</u> NONE (Parent class)

Basketball	
<u>Contains/Local:</u>	<u>Inherits from Team:</u>
<u>Contains/Local:</u> winPercentage() playSeason() toString()	<u>Inherits from Team:</u> getName() getWins() getLosses() setWins() setLosses()

BASEBALL	
<u>Contains/Local:</u>	<u>Inherits from Team:</u>
<u>Contains/Local:</u> winPercentage() playSeason() toString()	<u>Inherits from Team:</u> getName() getWins() getLosses() setWins() setLosses()

Before we break it down, let's observe a sample output from running the above program (it will be different each time due to Random objects being used):

Basketball team: Knicks
 Coached by: David Fizdale
 Awaiting season results

Baseball team: Mets
Coached by: Terry Collins
Awaiting season results

Basketball team: Heat
Coached by: Eric Spolstra
Awaiting season results

Basketball team: Knicks
Coached by: David Fizdale
Wins: 37
Losses: 45
Win percent: 0.45121951219512196
Losing season!

Baseball team: Mets
Coached by: Terry Collins
Wins: 85
Losses: 77
Win percent: 0.5246913580246914
Winning season!

Basketball team: Heat
Coached by: Eric Spolstra
Awaiting season results

(Okay, we know the Knicks stink, but does Java really need to rub it in? Moving on...)

The `main()` method will create three new classes of type `Basketball`, `Baseball`, and `Team`. It first prints out the information (via the use of the overloaded `toString()` methods we wrote) with the first three `System.out` statements. We have not played a season yet, so we do not have any information to show, hence the message “Awaiting season results” being printed as part of the output.

Next, we play the season for `team1` and `team2`. Why not `team3`? The answer is because it is of the parent class type. The parent class `Team` does not contain the `playSeason()` method. That method belongs to each of the child classes.

After we play the seasons for *team1* and *team2*, we reprint the information to the console. Since *team3* did not have a season played, the information at the end is correct.

EXAMPLE 4: *Multi-level Inheritance*

The below example will show multi-level inheritance, wherein one class inherits methods from another, that inherited methods from the parent class.

```
//A.java
public class A{
    //instance variables
    private int n1, n2;

    //default constructor setting variables to 0
    A() {
        n1 = n2 = 0;
    }

    //two argument constructor
    A(int n, int m) {
        n1 = n;
        n2 = m;
    }

    //get methods
    public int getN1() { return n1; }
    public int getN2() { return n2; }

    //set methods
    public void setN1(int n1) { this.n1 = n1; }
    public void setN2(int n2) { this.n2 = n2; }

    //member method
    public int mysteryA() {
        return n1 * n2;
    }
} //A class

//B.java
public class B extends A{
    //default constructor. Super constructor is A()
    B(){ super(); }

    //two argument constructor. Super constructor is A(n, m)
    B(int n, int m) {
        super(n, m);
    }

    //member method
}
```

```

public int mysteryB() {
    return getN1() + mysteryA();
}
} //B class

//C.java
public class C extends B{
    //default constructor. Super constructor is B()
    C(){ super(); }

    //two argument constructor. Super constructor is B(n, m)
    C(int n, int m) {
        super(n, m);
    }

    //member method
    public int mysteryC() {
        return getN2() + mysteryB();
    }
} //C class

//Example4.java
public class Example4{
    public static void main(String args[]){
        A a = new A();
        B b = new B(2, 2);
        C c = new C(5, 3);

        //print method returns
        System.out.println(a.mysteryA());
        System.out.println(b.mysteryB());
        System.out.println(c.mysteryC());

        //make some changes
        b.setN1(5);
        c.setN2(5);
        a.setN1(b.getN2());
        a.setN2(c.getN2());

        //print method returns
        System.out.println(a.mysteryA());
        System.out.println(b.mysteryB());
        System.out.println(c.mysteryC());
    } //main
} //class

```

Let's determine which methods are inherited, and which are local:

A

<u>Contains/Local:</u>	<u>Inherits:</u>
getN1() getN2() setN1() setN2() mysteryA()	NONE (Parent class)

B	
<u>Contains/Local:</u>	<u>Inherits from A:</u>
mysteryB()	getN1() getN2() setN1() setN2() mysteryA()

C	
<u>Contains/Local:</u>	<u>Inherits from B:</u>
mysteryC()	getN1() getN2() setN1() setN2() mysteryA() mysteryB()

In other words (for class C), it inherits the methods that class B inherited from class A.

The output from running the Example4 class is:

```
0
6
23
10
15
35
```

Let's breakdown why that is. Firstly, we create each object, giving it no initial value for *a*, but giving initial values for objects *b* and *c*. Currently, here is the breakdown of each variable in each object:

<i>a</i>	<i>n1</i> = 0	<i>n2</i> = 0
<i>b</i>	<i>n1</i> = 2	<i>n2</i> = 2
<i>c</i>	<i>n1</i> = 5	<i>n2</i> = 3

We then print out the returned result from *mysteryA()* of the *a* object. That method simply returns the product of *n1* and *n2*. Since they are both 0, it returns 0, and prints that value to the console.

Next, we call the *mysteryB()* method of the *b* object. Here, that method gets the current value of *n1*, and adds it to the result of the *mysteryA()* method it inherited. The current value of *n1* is 2. The *mysteryA()* method will return $2 * 2$, or 4. The final return value of *mysteryB()* is 6. That result is printed to the screen.

Next, we call the *mysteryC()* method of the *c* object. Here, that method gets the current value of *n2* and adds it to the result of the *mysteryB()* method it inherited from class B. The current value of *n2* is 3. The *mysteryB()* method will get the current value of *n1*, which is 5, and add it to the result of *mysteryA()*. The *mysteryA()* method returns $n1 * n2$, or $5 * 3$, which is 15. Going back to *mysteryB()*, it returns 15, plus the current value of *n1*, which is 5, for a result of 20. Now heading back to *mysteryC()*, it adds the current value of *n2* to the result of 20, for a value of 23. That result is printed to the screen.

We then make some changes to the value of some variables in each object, via the inherited *setN1()* or *setN2()* methods. The updated value of each variable is:

<i>a</i>	<i>n1</i> = 2	<i>n2</i> = 5
<i>b</i>	<i>n1</i> = 5	<i>n2</i> = 2
<i>c</i>	<i>n1</i> = 5	<i>n2</i> = 5

Rerun the trace through each method with the above current values to get the final three lines of output.

Dizzying yes, but this shows that classes inherit methods from classes that inherited methods.

EXERCISES

DIRECTIONS: Answer each question below to the best of your ability. Solutions on page 653.

Problem 1:

```
public class Math{
    private static int x = 0;

    public Math(int a){
        x = a;
    }

    public static int square(){ return x*x; }
    public static int cube(){ return x*x*x; }
    public static int pow(int e){
        if(e == 0) return 1;
        else if(e == 1) return x;

        int ans = x;
        for(int i = 2; i <= e; i++){
            ans *= x;
        }

        return ans;
    }

    public static int factorial(){
        int ans = 1;
        for(int i = 2; i <= x; i++){
            ans *= i;
        }

        return ans;
    }
}

public class Subject extends Math{
    public Subject(int a){
        super(a);
    }
    public static void main(String args[]){
        new Subject(5);
        int x = 0;

        x = factorial();
        System.out.println(x);
        System.out.println(square());
        System.out.println(cube());
        System.out.println(pow(3));
    }
}
```

```

        x = cube() + square() + factorial() + pow(2);

        System.out.println(x);
    }
}

```

1. What is the super class?
2. What methods are inherited from the super class?
3. What is the output from the above program?

Problem 2:

```

public class Team{
    private String name = "";
    private int members = 0;

    public Team(){
        name = "";
        members = 0;
    }
    public Team(String n, int m){
        name = n;
        members = m;
    }
    public String getName(){ return name; }
    public int getMembers(){ return members; }
    public void setName(String n){ name = n; }

    public void memberLeft(){ members --; }
    public void memberJoined(){ members ++; }

}

public class Baseball extends Team{
    public Baseball(){ super(); }
    public Baseball(String n, int m){
        super(n,m);
        setName("Mets");
        simulate();
    }

    public void simulate(){
        for(int i = 0; i < 7; i++){
            if(i % 4 == 1){
                System.out.println(getName() + " player left!");
                memberLeft();
            }else{
                System.out.println(getName() + " player joined!");
            }
        }
    }
}

```

```

        memberJoined();
    }
}
}

public static void main(String args[]){
    Baseball b = new Baseball("Yankees", 15);
    int total = b.getMembers() - 15;
    System.out.println("Total addition are: " + total);
}
}

```

1. What is the super class?
2. What methods are inherited from the super class?
3. What is the output from the above program?

Problem 3:

```

public class Fruit{
    private int quantity = 0;
    private double sales = 0.0;

    public Fruit(int q){
        quantity = q;
    }
    public boolean pick(){
        if(quantity == 0){
            System.out.println("Can't sell it!");
            return false;
        }
        quantity--;
        return true;
    }
    public void sell(double s){
        if(pick())
            sales += s;
    }
    public double getSales(){ return sales; }
}

public class Apple extends Fruit{
    double price = 0.0;

    public Apple(int q, double p, int t){
        super(q);
        price = p;
        store(t);
    }
    public void store(int times){

```

```

        for(int i = 0; i < times; i++) {
            sell(price);
        }
        System.out.println("Apples sold: $" + getSales());
    }

}

public class Orange extends Fruit{
    double price = 0.0;

    public Orange(int q, double p, int t){
        super(q);
        price = p;
        store(t);
    }
    public void store(int times){
        for(int i = 0; i < times; i++) {
            sell(price);
        }
        System.out.println("Oranges sold: $" + getSales());
    }
}

public class Main{
    public static void main(String args[]){
        Apple a = new Apple(6, 1.09, 4);
        Orange o = new Orange(2, 0.89, 3);
    }
}

```

1. What is the super class?
2. What methods are inherited from the super class?
3. What is the value of the *quantity* variable inside the Orange object *o* at the end of the program?
4. What is the output from the above program?

Problem 4:

```

public class Animal{
    private String name="";
    private int age = 0;
    public static int count = 0;

    public Animal(String n, int a, boolean b){
        count++;
        if(b == true && a == 5) {

```

```

        name = "JERRY";
        age = 100;
    }else{
        name = n;
        age = a;
    }
}

public int count(){ return count; }
public int age(){ return age; }
public String name(){ return name; }

public void setName(String n){ name = n; }

public String jumble(){
    char c[] = name.toCharArray();
    String ans = "";

    for(int i = 0; i < name.length()-1; i++){
        char temp = c[i];
        c[i] = c[i+1];
        c[i+1] = temp;
        ans += c[i];
    }

    return ans;
}
}

public class Dog extends Animal{
    public Dog(String s, int a, boolean b){
        super(s,a,b);
        setName(jumble());
    }
    public String toString(){
        return name() + " is " + age() + " years old!";
    }
}

public class Cat extends Animal{
    public Cat(String s, int a, boolean b){
        super(s,a,b);
        setName(jumble());
    }
    public String toString(){
        return name() + " is " + age() + " years old!";
    }
}

public class Main{
    public static void main(String[] args){

```

```

Animal animals[] = new Animal[5];
String names[] = {"Alice", "Steve", "Jake", "Max", "Ralph"};
int ages[] = {4,6,5,5,2};

for(int i = 0; i < 5; i++){
    if(i % 2 == 0)
        animals[i] = new Dog(names[i], ages[i], true);
    else
        animals[i] = new Cat(names[i], ages[i], false);
}

int length = animals[0].count();
for(int i = 0; i < length;
    System.out.println(animals[i]);
}
}

```

1. What are the two subclasses?
2. What is the purpose of the variable count?
3. How many instances of the class Dog are there?
4. What is the output from the above Main class?

Problem 5:

```

public class A{
    public int x = 0, y = 0;

    public A(int a, int b){
        x = a;
        y = b*a;
    }

    public int multiply(){ return x*y; }
    public int squareX(){ return x*x; }

    public String toString(){ return "A: x= " + x + " y= " + y; }
}

public class B extends A{
    private int xx = 0, yy = 0;

    public B(int aa, int bb){
        super(bb,aa);
        xx = aa;
        yy = bb;
    }
}

```

```

public int multiply(){ return xx * yy; }
public int mystery(){ return (squareX() + yy) * 2; }
public int mystery2(){ return x + y; }

public String toString(){ return "B: xx= " + xx
                        + " yy= " + yy;
}
}

public class Main{
    public static void main(String[] args){
        int a = 5, b = 3, c = 10;
        A[] arrA = new A[3];
        B[] arrB = new B[5];

        for(int i = 0; i < 5; i++){
            if(i < 3) arrA[i] = new A(b, c);
            arrB[i] = new B(c-i, a-i);
        }

        System.out.println(arrB[3].multiply());
        System.out.println(arrB[0].mystery2());

        for(int i = 0; i < 5; i++){
            if(i < 3) System.out.println(arrA[i]);
            System.out.println(arrB[i]);
        }
    }
}

```

1. What is the super class for the class B?
2. What methods are inherited to the class B?
3. What is the output from the class Main?

CHAPTER 15

Exceptions

This chapter will cover the basic ideas behind exceptions, which represent some kind of error when a program is run.

TOPICS

1. <i>Types of Exceptions</i>	339
2. <i>List of Exceptions</i>	339
3. <i>Try/Catch Blocks</i>	340
4. <i>Defining Exceptions</i>	345

In Java, an **exception** can be thought of as an error when a program is being run. There are numerous types of exceptions that may occur.

TYPES OF EXCEPTIONS

An exception can be **checked** or **unchecked**. If the exception is checked, you are able to recover from the mistake and possibly continue the program. An example of this may be a user entering the wrong name of location of a text file. A good program will catch the exception and ask the user to enter the name again. All of these checked exceptions do need to be caught (see try/catch below).

If the exception is unchecked, there is no need for a try/catch block. There are two subcategories of unchecked exceptions:

Runtime Exceptions

These occur when the program has a logical error or bug that you did not anticipate when programming. An example of a Runtime Exception can be the user not entering a file name, and causing the program to open a null file.

Error Exceptions

These will not be discussed here, but are caused by an error with the hardware of a computer.

LIST OF EXCEPTIONS

There are many definitions of exceptions in Java (too many, in fact, to list them all). Below are the most common types of exceptions that a programmer may see. Next to the exception is the type of exception it is:

IllegalArgumentException (Runtime)

An exception to handle a problem with method arguments or command-line arguments. One way to think of this is if the user does not give a needed command-line argument or the argument in a method is not in a certain range. This is the appropriate exception to be thrown.

NumberFormatException (Runtime)

This is an exception to handle a problem when formatting a number(s). This is commonly seen when using any of the parseXXX methods of a certain wrapper class. Say that your string is "1234ff" and you call

`parseInt`. When you get to the 'f', the exception will be thrown, since 'f' is not a number.

ArrayIndexOutOfBoundsException (Runtime)

The name says it all. This exception will occur when an index in an array is out of bounds in either direction (less than 0 or greater than or equal to its size).

NullPointerException (Runtime)

Ah yes, the most commonly seen and most annoying exception of them all! This means that you are pointing at nothing (or null) perhaps in a linked list or even when dealing with a `JOptionPane`. The solution: a bottle of Tylenol and good debugging skills.

IOException (Checked)

This is contained in the `java.io` library. It signals that some exception occurred during input/output of a program or reading or writing to a file.

FileNotFoundException (Checked)

This is also contained in the `java.io` library. It signals that a certain file cannot be found when the program attempted to open it.

NoSuchElementException (Runtime)

This is contained in the `java.util` library. This goes with the `StringTokenizer` object. It means that a certain element does not exist. What may cause this is a situation where there are 3 tokens and you tokenize more than that.

Exception (Checked)

The most general type of exceptions of them all. This will not specifically check for anything, but will mean some kind of error occurred. It will catch any kind of exception.

TRY/CATCH BLOCKS

In order for an exception to be caught, it must be used with a **try/catch** block. Below is the most general format of a try/catch block:

```
try{
    //code here
}catch(ExceptionName varName){
    //code to handle exception here
}
```

Where in the above, **catch** will contain the appropriate code to deal with an Exception. This may be where you print out the message or contain some other code to try and deal with the error. *ExceptionName* is the type of exception you are looking for (NoSuchElementException, NumberFormatException, etc.); and *varName* is the name of Exception object.

There is also a **try/catch/finally** block. This form is defined below:

```
try{
    //code here
}catch(ExceptionName varName){
    //code to handle exception here
}finally{
    //code here
}
```

What the *finally* section will do is contain some block of code that will execute, whether or not an exception is thrown. Say that there is an *IllegalArgumentException* thrown; the code for the catch block will execute, but immediately after that, the finally code will execute. This is one way of continuing a program.

There is also a global **Exception** object that will catch any type of exception when the program is run. That can be coded in a try/catch block or try/catch/finally block as such:

```
try{
    //code here
}catch(Exception varName){
    //code to handle exception here
}

try{
    //code here
}catch(Exception varName){
    //code to handle exception here
}finally{
    //code here
}
```

Some examples will help clarify exception handling.

EXAMPLE 1: Try/Catch Example

Below is a short example using a try/catch/finally block.

```
public class Example1{
    public static void main(String args[]){
        String str = "2346512aa";
        int sum = 0;

        for(int i = 0; i < str.length(); i++){
            try{
                sum += Integer.parseInt(str.charAt(i)+"");
            }catch(NumberFormatException nfe){
                System.out.print("Yikes!  ");
            }finally{
                System.out.println("i= " + i);
            }
        }
        System.out.println("Sum: " + sum);
    } //main
} //class
```

The above program will attempt to add each digit in a numeric string. We want to attempt to catch a *NumberFormatException*. When the program hits the first non-numeric character in the String, the *NumberFormatException* will be thrown and be caught. No matter how many times the exception is thrown and caught, the finally code will execute, which simply prints the loop counter.

The output from running the above program is:

```
i= 0
i= 1
i= 2
i= 3
i= 4
i= 5
i= 6
Yikes! i= 7
Yikes! i= 8
Sum: 23
```

Let's say that you had more than one catch block for the above program. The kind of exception you catch depends on where you place the catch block.

EXAMPLE 2: Try/Catch Example II

Here is a modified example of the above program.

```
public class Example2{
    public static void main(String args[]){
        String str = "2346512aa";
        int sum = 0;

        for(int i = 0; i <= str.length(); i++){
            try{
                sum += Integer.parseInt(str.charAt(i) + "");
            }catch(IndexOutOfBoundsException ioe){
                System.out.print("Regular!  ");
            }catch(NumberFormatException nfe){
                System.out.print("Yikes!  ");
            }finally{
                System.out.println("i= " + i);
            }
        }
        System.out.println("Sum: " + sum);
    } //main
} //class
```

The output is now:

```
i= 0
i= 1
i= 2
i= 3
i= 4
i= 5
i= 6
Yikes! I= 7
Yikes! I= 8
Regular! I= 9
Sum: 23
```

Because of the newly-placed *IndexOutOfBoundsException*, the *NumberFormatException* is checked last, while the *IndexOutOfBoundsException* is checked first.

The order of the exceptions is read from top to bottom, **in order**.

EXAMPLE 3: *Global Exception*

Here is another modified example of the above program.

```
public class Example3{
    public static void main(String args[]){
        String str = "2346512aa";
        int sum = 0;

        for(int i = 0; i <= str.length(); i++){
            try{
                sum += Integer.parseInt(str.charAt(i) + "");
            }catch(Exception e){
                System.out.println(e);
            }finally{
                System.out.println("i= " + i);
            }
        }
        System.out.println("Sum: " + sum);
    } //main
} //class
```

The output is now:

```
i= 0
i= 1
i= 2
i= 3
i= 4
i= 5
i= 6
java.lang.NumberFormatException: For input string: "a"
i= 7
java.lang.NumberFormatException: For input string: "a"
i= 8
java.lang.StringIndexOutOfBoundsException: String index out of range: 9
```

i= 9

Sum: 23

As seen with the above output, when you print the type of exception to the console, you see what type of exception is thrown, and a brief message as to what the error is.

DEFINING EXCEPTIONS

Using the power of inheritance, you can create your own kind of exceptions. Say that you have a program that relates to both Graduate and Undergraduate students in college. More specifically, the names of the students are being read from a text file.

Instead of throwing a general *IllegalArgumentException*, which may apply to either the Graduate or Undergraduate classes, you may want to create a useful exception called *GradStudentException* or *UnderGradStudentException*; both of which can be subclasses of the *IllegalArgumentException*.

Below is the definition, in general, of your own exceptions:

```
public class NewExceptionName extends CurrExceptionName{  
    public NewExceptionName(String gripe){  
        super(gripe); //super class: CurrExceptionName  
    }  
}
```

Where in the above, *NewExceptionName* is your own name for an exception (such as *GradStudentException*); and *CurrExceptionName* is the name of a current exception (such as *IllegalArgumentException*). There is also the constructor for the new exception to allow a String argument to describe the problem. This is passed to the super class.

Below is a new exception in regard to the student program before:

```
public class GradStudentException extends IllegalArgumentException{  
    public GradStudentException(String gripe){  
        super(gripe);  
    }  
}
```

The file name for each new exception will be *NewExceptionName.java* and placed in the same directory as the main program execution.

In order for you to trigger a certain exception, you must use the **throw** keyword.

EXAMPLE 4: *Custom Exceptions*

The below program will ask the user for a month, day, and year entry (all int values). It then performs some error checking on the entries when trying to create the DateEntered object. If a particular error is caught, a respective Exception is thrown. These exceptions will be defined to help us narrow down what type of error occurred when the user entered the data. If all is correct, then the program outputs the entries in date form.

A leap year is defined as a year that is evenly divisible by 4, unless it falls on a century. If it falls on a century, the year must be evenly divisible by 400. An example is 2000. That is a century year and is evenly divisible by 400. An example of a century year that is not a leap year is 1900. While it is divisible by 4 and is a century, it is not evenly divisible by 400.

```
//MonthException.java
public class MonthException extends IllegalArgumentException{
    MonthException(String g){ super(g); }
}

//DayException.java
public class DayException extends IllegalArgumentException{
    DayException(String g){ super(g); }
}

//YearException.java
public class YearException extends IllegalArgumentException{
    YearException(String g){ super(g); }
}

//FebruaryException.java
public class FebruaryException extends IllegalArgumentException{
    FebruaryException(String g){ super(g); }
}

//DateEntry.java
public class DateEntry{
    private int month, day, year;

    DateEntry(int m, int d, int y){
```

```

//check for correct month
if(m < 1 || m > 12){
    throw new MonthException("Month value out of range");
}

//for this program, user can enter between 1900 and 2019
if(y < 1900 || y > 2019) {
    throw new YearException("Year value out of range");
}

//general day check
if(d < 1)
    throw new DayException("Month needs at least 1 day");

//check for the proper number of days
switch (m) {
//30 days as September, April, June and November
case 9:
case 4:
case 6:
case 11:
    if(d > 30)
        throw new DayException(
            "Month can't have more than 30 days");
    break;

//February so check for a leap year
case 2:
    boolean isLeap = false;

    //checking for leap year
    if(y % 4 == 0) {
        //is the year a century year (such as 1900)
        //if not, simple leap year
        if(y % 100 == 0) {
            //if century year is also divisible by 400
            if(y % 400 == 0) {
                isLeap = true;
            }else {
                isLeap = false;
            }
        }else {
            //regular leap year such as 2004
            isLeap = true;
        }
    }

    //leap years can't have more than 29 days
    if(isLeap && d > 29)
        throw new FebruaryException(
            "February can't have more than 29 days in a leap year");
    else if(!isLeap) && d > 28)
}

```

```

                throw new FebruaryException(
"February can't have more than 28 days in a non-leap year");
        }else {
            if(d > 28)
                throw new FebruaryException(
"February can't have more than 28 days in a non-leap year");
        }
        break;

//rest of calendar months
default:
    if(d > 31)
        throw new DayException(
            "Month can't have more than 31 days");
    break;
}
//otherwise, good on entries
month = m;
day = d;
year = y;
}

//overloaded toString() method
public String toString() {
    return "You entered: " + month + "/" + day + "/" + year;
}
}

//Example4.java
public class Example4{
    public static void main(String args[]){
        //Scanner and variables
        Scanner s = new Scanner(System.in);
        int m = 0, d = 0, y = 0;
        try {
            //prompt user for name entry
            System.out.println("Enter month (1-12): ");
            m = s.nextInt();

            //prompt user for name entry
            System.out.println("Enter day (1-31): ");
            d = s.nextInt();

            //prompt user for name entry
            System.out.println("Enter year (1900-2019): ");
            y = s.nextInt();

            //try to build new DateEntry object
            DateEntry date = new DateEntry(m, d, y);

            System.out.println(date);
        }
    }
}

```

```
        }catch(Exception e) {
            System.out.println(e);
        }
    } //main
} //class
```

Let's run the program with a few sample trials.

Enter month (1-12):

1

Enter day (1-31):

2

Enter year (1900-2019):

1905

You entered: 1/2/1905

This is a proper run. All ranges are acceptable. Let's try another that's out of range:

Enter month (1-12):

13

Enter day (1-31):

2

Enter year (1900-2019):

1900

MonthException: Month value out of range

The program does correctly catch the MonthException we built. Here is another run:

Enter month (1-12):

4

Enter day (1-31):

33

Enter year (1900-2019):

1902

DayException: Month can't have more than 30 days

April, by the rules of the calendar, cannot have more than 30 days, so the program does successfully catch the error. Here is another run:

```
Enter month (1-12):  
3  
Enter day (1-31):  
17  
Enter year (1900-2019):  
2020  
YearException: Year value out of range
```

By the rules of our program, the year needs to be between 1900 and 2019. The program successfully catches the exception. Here is another run:

```
Enter month (1-12):  
2  
Enter day (1-31):  
29  
Enter year (1900-2019):  
1900  
FebruaryException: February can't have more than 28 days in a non-leap year
```

The program checks if the year is a leap year. Here, 1900 is not a leap year (as per the leap year algorithm). so this program does successfully catch the error. Here is another run for February:

```
Enter month (1-12):  
2  
Enter day (1-31):  
29  
Enter year (1900-2019):  
1904  
You entered: 2/29/1904
```

The year 1904 is a leap year, so the program runs successfully.

CHAPTER 16

File Input / Output

This chapter discusses the topic of file input and output in Java. Shows not only how to both read from a file & process data, but also how to write to a file.

TOPICS

1. <i>File Input</i>	352
2. <i>File Output</i>	357
3. <i>Exercises</i>	365

In Java, file input and file output are done with a **BufferedReader** (input) and **BufferedWriter** (output). Each of them is a stream, just like the C++ <fstream>.

FILE INPUT

In order for you to have a BufferedReader object, you must import both of the below:

```
import java.io.BufferedReader;
import java.io.FileReader;
```

After that, you can declare the BufferedReader object as follows:

```
BufferedReader varName = new BufferedReader(
    new FileReader(filename));
```

This will then instantiate the object and allow you to read a line of text. Note that inside the constructor of the BufferedReader is another object called FileReader. **BOTH** of these need to be instantiated in order for the entire BufferedReader object to be created.

Most often paired with a BufferedReader (though not always), is a StringTokenizer that will allow you to read an entire line of a file, and break it up into pieces (if that is needed). The StringTokenizer object is in the java.util library (see the String chapter for a refresher on the StringTokenizer).

There is one golden rule about file input or output: the object **MUST** be declared inside a **try/catch** block; there is no exception to this rule.

Let's review some of the methods that a BufferedReader will use:

String readLine()

This will extract an entire line of text as denoted by the carriage return character '\r' or a new line character '\n'. The return type is String because it will extract a String from the file.

int read()

There is also another method in the class called read(). This will simply read an individual character. It will return a -1 if the end of the stream has been reached.

```
void close()
This will simply close the input stream.
```

EXAMPLE 1: *BufferedReader* for Input

Here is short example that will show a BufferedReader with a text file called "example1.txt" that looks like the following:

1,10,5,3,12,16,100,56,74,23,1,2,3,5,6,77

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.util.StringTokenizer;

public class Example1{
    public static void main(String args[]){
        StringTokenizer st;

        try{
            BufferedReader br = new BufferedReader(
                new FileReader("example1.txt"));

            //get line of the file
            String line = br.readLine();

            //close the input stream
            br.close();

            double avg = 0.0;
            int num = 0, count = 0;

            //default token is white space
            st = new StringTokenizer(line, ",");

            for(int i = 0; i < st.countTokens(); i++){
                avg += Double.parseDouble(st.nextToken());
                count++;
            }

            //calculate and print average
            avg = avg/count;
            System.out.println("Avg: " + avg);

        }catch(Exception e){}
    }
}
```

```
    } //main
} //class
```

This program will read a file of integers, and calculate the average. It will parse each integer into a double value, since the average itself may include a decimal number. The output from above is: "Avg: 24.625"

EXAMPLE 2: *Mean, Min and Max*

The tasks of the following program are as follows:

1. Read a set of numbers from an input file, each separated by a comma.
2. Place those numbers in an array of integers.
3. Calculate the mean (average), min and max values of the numbers given. You are **NOT allowed** to make use of the java Math library.

Let's break the above tasks down into smaller pieces. Firstly, we know that we need a BufferedReader for the input in part 1. We also know we need a StringTokenizer for the data itself, since it is separated by commas. Lastly, we should write our own methods called *min()*, *max()* and *mean()* of the appropriate return types.

So, here is a sample input file (example2.txt) consisting of 20 integers:

```
1,5,23,6,7,76,3,2,4,5,7,8,56,3,2,1,1,3,45,99
```

Here is the program written out in full. An explanation will follow.

```
import java.util.StringTokenizer;
import java.io.BufferedReader;
import java.io.FileReader;

public class Example2{
    //method to calculate the min value in an array
    private static int min(int[] n){
        int m = n[0];
        for(int i = 1; i < n.length; i++)
            if(n[i] < m) m = n[i];
        return m;
    } //min

    //method to calculate the max value in an array
    private static int max(int[] n){
        int m = n[0];
```

```

        for(int i = 1; i < n.length; i++)
            if(n[i] > m) m = n[i];
        return m;
    } //max

    //method to calculate the mean value of an array
    private static double mean(int[] n) {
        int sum = 0;
        for(int i = 0; i < n.length; i++)
            sum += n[i];

        return (sum / n.length );
    } //mean

    public static void main(String args[]){
        try{
            //declare the file for reading
            BufferedReader br = new BufferedReader(
                new FileReader("example2.txt"));

            String line = br.readLine(); //get the line
            br.close(); //close the file

            //declare the tokenizer
            StringTokenizer st = new StringTokenizer(line, ",");

            //array size is the number of tokens in the file
            int[] nums = new int[st.countTokens()];

            //place the numbers in the array
            for(int i = 0; i < nums.length; i++){
                nums[i] = Integer.parseInt(st.nextToken());
            }

            //now call methods
            System.out.println("Min: " + min(nums));
            System.out.println("Max: " + max(nums));
            System.out.println("Mean: " + mean(nums));
        }catch(Exception e){
            System.out.println("whoops!" + e);
        }
    } //main
} //class

```

The program first creates the BufferedReader object, giving it the name of the input file. It then reads the line and closes the stream right away, since we are only dealing with one line of input. The token for the line is a comma as seen above.

The array is created dynamically. In this case, it counts the number of tokens in the line of text you extracted, and declares an array of that size (in this case, 20). It fills the array with the integers by parsing them.

Once all the housekeeping is finished, we can now call each method to carry out the calculations we need.

The output of the above program is:

```
Min: 1
Max: 99
Mean: 17.85
```

EXAMPLE 3: Eliminate Consecutive Numbers

The below program will:

1. Read a line of numbers from an input file, each separated by a whitespace.
2. Filter out consecutive numbers in the file. A consecutive number is defined as the current number you are observing being the same as its neighbor.
3. Print to the screen a final string of non-consecutive numbers.

The input file will look like:

```
1 2 2 2 1 4 3 3 5 6 5 5 3 4 3 2 2 2 1 2
```

The final result of the program should yield:

```
1 2 1 4 3 5 6 5 3 4 3 2 1 2
```

```
import java.util.StringTokenizer;
import java.io.BufferedReader;
import java.io.FileReader;
public class Example3{
    public static void main(String args[]) {
        try {
            BufferedReader br = new BufferedReader(
                new FileReader("example3.txt"));

            //read line from file and close stream
            String line = br.readLine();
            br.close();
```

```

//tokenizer for the line (default token white space)
StringTokenizer st = new StringTokenizer(line);

String curr = "", temp = "";
curr = st.nextToken(); //read first number
System.out.print(curr + " "); //print first number

//keep program running until there are no more tokens
while(true) {
    //get next number
    temp = st.nextToken();

    //reached the last token so break out of loop
    if(!st.hasMoreTokens()) break;

    //if the current number does not equal the next
    //number, you have found a unique so adjust
    //curr variable and print
    if(!curr.equals(temp)) {
        curr = temp;
        System.out.print(temp + " ");
    }
}
//make final comparison here
if(!curr.equals(temp)) {
    System.out.print(temp + " ");
}

}catch(Exception e) {
    System.out.println(e);
}
} //main
} //class

```

The output from running the above program produces the desired result:

1 2 1 4 3 5 6 5 3 4 3 2 1 2

Feel free to adjust the numbers in the input file to experiment.

FILE OUTPUT

Just the opposite of the BufferedReader is the **BufferedWriter**, which will write to a text file. In order for you to have a BufferedWriter object, you must import both of the below:

```
import java.io.BufferedWriter;
import java.io.FileWriter;
```

Here is how to declare the BufferedWriter object:

```
BufferedWriter varName = new BufferedWriter(
    new FileWriter(filename));
```

For the BufferedWriter class, if the output file is not currently in the directory, it will create the file for you. If the file is already there, the output will overwrite the information in the output file. Just like the BufferedReader, the BufferedWriter **must be** contained in a try/catch block.

Let's see some useful methods of the BufferedWriter object:

void write(String s)
void write(char c)

This will write a String argument or a single character to the destination file.

void newLine()

This will write a line separator to the file. This is not quite the new line character, but it can be thought of that way.

void close()

This will simply close the output stream.

EXAMPLE 4: *BufferedWriter for Output*

Here is a short program that will take, as a command line argument, a sentence (that will need quotes), and output to the file a triangle of characters.

```
import java.io.BufferedWriter;
import java.io.FileWriter;
public class Example4{
    public static void main(String args[]){
        //check for bad argument
        if(args.length == 0){
```

```

        System.out.println("String argument needed.");
        System.exit(1);
    }

    //get the entered argument
    String word = args[0];

    try{
        BufferedWriter bw = new BufferedWriter(
            new FileWriter("output.txt"));

        String ans = "";

        //loop through the String entered and print the characters
        for(int i = 0; i < word.length(); i++){
            ans += word.substring(i);
            bw.write(ans);
            bw.newLine();
            ans = "";
        }
        //close the output stream
        bw.close();

    }catch(Exception e){}
} //main
} //class

```

For the purpose of this example, the String “hello there” will be used. The output from running the above program, to the text file, is:

```

hello there
ello there
llo there
lo there
o there
there
there
here
ere
re
e

```

EXAMPLE 5: Dice Rolls Revisited

In Chapter 11, we learned about the Random number library. Recall the program we wrote for counting the number of times a user rolls a double when rolling dice:

```
import java.util.Random;
public class Example5{
    public static void main(String args[]) {
        Random d1, d2;

        //initialize new Random objects representing
        //two dice
        d1 = new Random();
        d2 = new Random();

        //array to track the doubles, allows for using index 1 to 6
        int pairs[] = new int[7];
        int d1_roll = 0, d2_roll = 0;

        //roll the pair 1000 times
        for(int i = 0; i < 1000; i++) {
            //roll the pair of dice (numbers 1 to 6)
            d1_roll = d1.nextInt(6) + 1;
            d2_roll = d2.nextInt(6) + 1;

            //same number was rolled
            if(d1_roll == d2_roll)
                pairs[d1_roll]++;
        }

        //print the results to the user
        for(int i = 1; i <= 6; i++)
            System.out.println("Pairs of " + i + "s were rolled "
                               + pairs[i] + " times.");
    } //main
} //class
```

Now let's actually run the program and write, to a text file, the results of the actual dice rolls. We will still print the results of the doubles to the screen.

```
import java.io.BufferedReader;
import java.io.FileWriter;
import java.util.Random;
public class Example5{
    public static void main(String args[]) {
        try {
            BufferedWriter bw = new BufferedWriter(
                new FileWriter("output.txt"));
```

```

Random d1, d2;

//initialize new Random objects representing
//two dice
d1 = new Random();
d2 = new Random();

//array to track doubles, allows for using index 1-6
int pairs[] = new int[7];
int d1_roll = 0, d2_roll = 0;

//roll the pair 1000 times
for(int i = 0; i < 1000; i++) {
    //roll the pair of dice (numbers 1 to 6)
    d1_roll = d1.nextInt(6) + 1;
    d2_roll = d2.nextInt(6) + 1;

    //write the two numbers
    bw.write(d1_roll + " - " + d2_roll);
    bw.newLine(); //go to next line

    //same number was rolled
    if(d1_roll == d2_roll)
        pairs[d1_roll]++;
}

//close input stream
bw.close();

//print the results to the user
for(int i = 1; i <= 6; i++)
    System.out.println("Pairs of " + i
        + "s were rolled " + pairs[i] + " times.");

} catch(Exception e) {
    System.out.println(e);
}
} //main
} //class

```

The output to the screen can be:

Pairs of 1s were rolled 23 times.
 Pairs of 2s were rolled 24 times.
 Pairs of 3s were rolled 33 times.
 Pairs of 4s were rolled 28 times.
 Pairs of 5s were rolled 36 times.

Pairs of 6s were rolled 32 times.

However, you will see a rather lengthy output file, constituting of each random roll of the dice. Run the program multiple times to generate different output.

EXAMPLE 6: *Student Test Scores*

We want to write a program that performs the following tasks:

- 1) Read from an input file consisting of 5 lines of data (see below)
- 2) Extract the name and grade information for each student
- 3) Calculate the highest, lowest and average test scores for each student (there are 10 test scores per student).
- 4) Write the results to a text file, not to the console.

The text file will consist of the following (formatted exactly as seen)

```
Alex|90,92,82,90,76,91,55,87,66,84  
James|100,92,81,57,47,91,100,87,100,90  
Frank|92,90,80,91,72,92,52,82,62,82  
Lisa|100,100,100,99,98,99,55,55,22,100  
Shannon|60,22,100,99,97,96,43,99,99,87
```

```
import java.io.*;  
import java.util.StringTokenizer;  
public class Example6{  
    public static void main(String args[]) {  
        try {  
            //declare the file for reading  
            BufferedReader br = new BufferedReader(  
                new FileReader("input2.txt"));  
  
            //declare the file for writing  
            BufferedWriter bw = new BufferedWriter(  
                new FileWriter("output.txt"));  
  
            //needed variables  
            String line = "", line2 = "", name = "";  
            String names[] = new String[5];  
            double temp[] = new double[10];  
  
            line = br.readLine(); //get first line  
            StringTokenizer st;  
  
            //loop through each students info  
            for(int i = 0; i < 5; i++) {
```

```

        //both tokens are needed. The | token separates
        //the name from the grades, while the comma
        //separates each grade
        st = new StringTokenizer(line, "|,");

        //get the name
        name = st.nextToken();

        //get the 10 grades and parse to a double
        for(int j = 0; j < 10; j++) {
            temp[j] =
                Double.parseDouble(st.nextToken());
        }

        //begin outputting the needed information
        line2 = name + " grade information:";
        bw.write(line2);
        bw.newLine();

        //find the average grade
        line2= "Average grade: " + findAvg(temp) + ".";
        bw.write(line2);
        bw.newLine();

        //find the highest grade
        line2= "Highest grade: " + findMax(temp) + ".";
        bw.write(line2);
        bw.newLine();

        //find the lowest grade
        line2= "Lowest grade: " + findMin(temp) + ".";
        bw.write(line2);
        bw.newLine();
        bw.newLine();

        //go to the next line
        line = br.readLine();
    }

    //close both input and output streams
    br.close();
    bw.close();
} catch(Exception e) {
    System.out.println(e);
}
} //main

//method to find the average grade in the given array
public static double findAvg(double arr[]) {
    double avg = 0.0;
    for(int j = 0; j < 10; j++) {

```

```

        avg += arr[j];
    }
    avg /= 10;
    return avg;
}

//method to find the highest grade in the given array
public static double findMax(double arr[]) {
    int index_max = 0;
    for (int i = 1; i < 10; i++) {
        if(arr[i] >= arr[index_max])
            index_max = i;
    }
    return arr[index_max];
}

//method to find the lowest grade in the given array
public static double findMin(double arr[]) {
    int index_min = 0;
    for (int i = 1; i < 10; i++) {
        if(arr[i] <= arr[index_min])
            index_min = i;
    }
    return arr[index_min];
}
} //class

```

When run, the output file will look like:

Alex grade information:

Average grade: 81.3.

Highest grade: 92.0.

Lowest grade: 55.0.

James grade information:

Average grade: 84.5.

Highest grade: 100.0.

Lowest grade: 47.0.

Frank grade information:

Average grade: 79.5.

Highest grade: 92.0.

Lowest grade: 52.0.

Lisa grade information:

Average grade: 82.8.

Highest grade: 100.0.

Lowest grade: 22.0.

Shannon grade information:

Average grade: 80.2.

Highest grade: 100.0.

Lowest grade: 22.0.

EXERCISES

DIRECTIONS: Answer each question below to the best of your ability. Solutions on page 654.

Problem 1:

Briefly describe each of the following as it relates to either file input or output:

- a) import java.io.BufferedReader
- b) import java.io.*
- c) close()
- d) readLine()
- e) writeLine()

Problem 2:

Write a program to read a file of integers called “input.txt” Count and display how many of them are prime numbers. It may help to know that a prime number is only divisible by 1 and itself. The input file will contain the following values and look exactly like this having an integer appear next to each other:

3 2 8 5 6 33 21 7 9 10

Problem 3:

Write a program to read a file of integers called “input.txt” Count and display how many of them are Fibonacci numbers (see Chapter 9 for a description of Fibonacci numbers). The file will be the same file as problem 2 above.

Problem 4:

Write a program to read a file of words called “words.txt” and write them to a file called “abc.txt” in alphabetical order. The input file will contain the following values and look exactly like this having a word appear on each line:

Mat
Car
Hat
Bat
Fat
Balloon
Garden
Sunshine
Zoo
Ready

Additional Problems

DIRECTIONS: Answer each problem below to the best of your ability. Solutions are not provided in this book.

Problem 5:

Write a program to perform the following tasks:

1. Reads an input file of email addresses called "emails.txt."
2. Place each valid email address in an array. Here is what defines valid:
 - a. -an @ symbol followed by some text and a period. If either of these fail, the address is not a valid address.
3. Counts and displays the number of valid users that have a gmail or yahoo account.

The input file will look like this:

```
allison@yahoo.com
youth_camp@gmail.com
abc123aol.com
123abc@aolcom
bpope1@hotmail.com
harryFords@hotmail.com
msimspsons@hotmailcom
```

Problem 6:

Write a program to perform the following tasks:

1. Have the user enter an integer n, which will represent the size of a 1 dimensional array of short integers.
2. Declare the array and read the first n integers from an input file called "input.txt" containing, at most 100 integers, 1 per line.
3. Find and display the average of the integers read with the use of a method.
4. Find and display the product of all the integers read with the use of a method.

Problem 7:

Write a program to perform the following tasks:

1. Create a class called *Student*, that will contain data for a student's name, ID number, Major and GPA. The class will have appropriate get methods for each instance variable.
2. Create a class called *Professor* that will contain the main method.
3. In the main method, perform the following tasks:
 - a. Read from an input file called "students.txt", which will extract the student data in each row. (See below for file formatting).
 - b. When processing the data, you will perform some error checking:
 - i. ID number must be exactly 6 digits in length. If it is longer than 6 digits, truncate the number to take the last 6 digits to make the ID. If it is shorter, pad 9s in the front to create the ID.
 - ii. Major can be only "Computer Science", "Mathematics," "Biology" or "English." If any other type exists, the value of the instance variable for major will be "UNKNOWN" in all uppercase.
 - c. Place each row of student data into an array of Student objects. The size is determine by reading the first line of the file, a single integer representing the size.
 - d. The program then outputs to the console the following:
 - i. Find the students with the highest and lowest GPAs. Print their name and current GPA to the 100th (3.14 for example).
 - ii. Display the total count of each major for the students.
 - iii. Display the names of the students that begin with a vowel (A, E, I, O or U).

The file will look exactly like the following:

10
Ryan Jones|104567|English|3,A|4,B+|3,C|5,A-|3,C+
Lisa Powers|167|Mathematics|3,F|3,B|3,C|3,D|3,D+
Andrew Maxwell|342347123833|Chemistry|4,A|2,B+|3,C+|4,A|3,F
Stephanie Anderson|045116721|Computer Science|4,A|4,B-|3,C-|5,D+|3,B
Charles Cochran|10456721234|Mathematics|3,A|4,B-|3,C|2,A|3,F
James Smith|111|Spanish|4,A+|4,B+|3,C-|4,A-|3,D
Adam West|6777|Biology|3,F|4,F|3,F|3,C-|3,F
Frank Scott|5267|History|3,C|3,B+|3,C|3,A-|4,B+
John Johnson|344334|Computer Science|3,A-|4,B-|3,C-|5,A-|3,C-
Edward Williams|909092|Mathematics|3,A+|3,B+|4,C+|3,D+|5,C+

To calculate a GPA, you take the grade points for a particular letter grade multiplied by the credits attempted for the course. The grade points are defined below:

Course Grade	Grade Points
A/A+	4.0
A-	3.7
B+	3.3
B	3.0
B-	2.7
C+	2.3
C	2.0
C-	1.7
D+	1.3
D	1.0
D-	0.7
F	0.0

To calculate the entire (or cumulative) GPA, you take the total number of grade points calculated divided by the total number of credit hours attempted.

Example for first student above, Ryan Jones:

Credit Hours	Course Grade	Grade Points
3	A	12
4	B+	13.2
3	C	6
5	A-	18.5
3	C+	6.9
18	-----	56.6

$$\text{GPA: } 56.6 / 18 = 3.14$$

CHAPTER 17

Abstract Classes & Interfaces

This chapter will cover the basic idea behind an abstract class (a class that cannot directly be constructed) and interfaces, often times used alongside abstract classes.

TOPICS

1. <i>Abstract Classes</i>	370
2. <i>Abstract Methods</i>	373
3. <i>Account Class</i>	373
4. <i>Shape Class</i>	383
5. <i>Interfaces</i>	389

When dealing with inheritance, we have seen the relationship between super classes and subclasses. However, this was all done on a public level.

An abstract, by its textbook definition, is a rough sketch of something. By its application in Java, an **abstract class** is a class that is a sketch of something to come. It is a class that **cannot be directly constructed**.

ABSTRACT CLASSES

In order for a class to be declared abstract, you must use the **abstract** keyword:

```
abstract class Name {  
    //code  
}
```

As an example, let's say that you are thinking of running an electronic store. We know that there can be CDs, DVDs, VHS tapes or even, for those oldies, Records. All of these are of a "super" type called Media. So for Java, the Media class can be declared abstract, while the other classes can be subclasses of Media. Here is a diagram showing this:



where Media would be abstract (in addition to the super class) and the other classes would extend Media.

EXAMPLE 1: Abstract Classes

Let's see that sketch above implemented in some code. Try to follow the program and compute the output before looking.

```
//MediaItem.java  
public abstract class MediaItem{  
    private String title; //global title variable  
    private int year, quantity; //global year & quantity variables  
  
    public MediaItem(String t, int yr, int q){  
        if(t == null)
```

```

        throw new IllegalArgumentException("No title given");
    else if(yr < 1900 || yr > 2019)
        throw new IllegalArgumentException("Bad year");
    else if(q < 0)
        throw new IllegalArgumentException("Bad quantity");
    else{
        title = t;
        year = yr;
        quantity = q;
    }
} //constructor

public String toString(){
    return("Title: " + title
        + "\nYear: " + year
        + "\nQuantity: " + quantity);
} //toString()
} //class

//CD.java
public class CD extends MediaItem{
    public CD(String t, int yr, int q){
        super(t, yr, q);
    } //constructor
} //class

//DVD.java
public class DVD extends MediaItem{
    public DVD(String t, int yr, int q){
        super(t, yr, q);
    } //constructor
} //class

//VHS.java
public class VHS extends MediaItem{
    public VHS(String t, int yr, int q){
        super(t, yr, q);
    } //constructor
} //class

//Store.java
public class Store{
    public static void main(String args[]){
        MediaItem items[] = new MediaItem[5];

        for(int i = 0; i < 5; i++){
            try{
                if(i % 3 == 0)
                    items[i] = new CD("Im a CD", 1993, i*3);
                else if(i % 3 == 1)

```

```

        items[i] = new DVD("Im a DVD", 1999, i*2);
    else
        items[i] = new VHS("Im old!", 1987, i*4);
    }catch(InvalidArgumentException iae){}
}

System.out.println("Inventory:");
for(int i = 0; i < 5; i++)
    System.out.println("Item: " + items[i] + "\n");
} //main
} //class

```

The above is an example to show how a class can be declared abstract. Each of the CD, DVD, and VHS classes are subclasses of *MediaItem*.

In the *main()* method of the class *Store*, we can declare an array of *MediaItem* objects. Here we declare it to be of size 5. This is certainly allowed so long as we don't say:

```
items[i] = new MediaItem( ... );
```

By the rules of abstract classes, this is not allowed. The abstract class *MediaItem* cannot be directly constructed. Also contained in the *main()* method of class *Store*, we include a try/catch block, since in the *MediaItem* constructor we are throwing an exception if a certain condition is met. This try/catch block is there just for good practice, since none of the items in this example are throwing an exception.

The output from running the above example is:

Inventory:

Item: Title: Im a CD

Year: 1993

Quantity: 0

Item: Title: Im a DVD

Year: 1999

Quantity: 2

Item: Title: Im old!

Year: 1987

Quantity: 8

Item: Title: I'm a CD

Year: 1993

Quantity: 9

Item: Title: I'm a DVD

Year: 1999

Quantity: 8

ABSTRACT METHODS

In Java, a method can be declared abstract. This means that it will not have an implementation within the abstract class in which it is being defined, but it will be implemented in a subclass. The method will use the same abstract keyword in its title:

```
[identifier] abstract type Name() ;
```

Where in the above, *identifier* can be public, private, or protected; *type* is the return type of the method; and *Name* is the name of the useful name of the method. Notice that there is a semicolon appearing at the end of the method declaration. This is how to denote an abstract method, since there is no implementation after it.

ACCOUNT CLASS

Let's use a real world example. In banking, there are two types of accounts, a checking and a savings. In reality, there are many more than that, but for the purpose of this example, we are using those two. In Java, we can declare an abstract class named *Account*, with a subclass of both *Checking* and *Savings*. As we've done in prior chapters, let's build this class piece by piece.

An account holder can withdraw, deposit or transfer funds from one account to another.

At the minimum, the abstract class will look like this:

```

public abstract class Account{
    private double balance;

    //constructor, taking the amount of the opened account
    public Account(double amt) {
        balance = amt;
    }

    //get/set methods
    public double getBalance() { return balance; }
    public void setBalance(double amt) { balance = amt; }

    //abstract methods for withdrawing and depositing to an account
    public abstract void withdraw(double amt);
    public abstract void deposit(double amt);
} //Account

```

It contains an instance variable for the balance of a respective account. The constructor will create the object with the given balance. There are also get/set methods for the balance, and abstract methods for *withdraw()* and *deposit()*. Those methods will be defined in the respective subclass of either *Checking* or *Savings*.

Checking Class

Continuing on, we want to create the Checking subclass:

```

public class Checking extends Account{
    //one argument constructor
    public Checking(double amt) {
        super(amt);
    }
    //overloaded toString() method which prints current balance
    public String toString() {
        return "Current balance in your " +
            this.getClass().getName() + " account is: $" +
            getBalance();
    }
}

```

So far, we have the constructor for the class, which takes the amount of the account and calls upon the super class's constructor. We also have overloaded the *toString()* method, which will obtain the balance of the account and print an appropriate message.

Next, we want to add the method to withdraw from the checking account:

```
//method to withdraw funds from the Checking account
public void withdraw(double amt) {
    //get the current balance via inherited
    //getBalance() method
    double bal = getBalance();

    //if the current balance minus the requested
    //withdrawal amount will produce a negative value
    //print message and exit method
    if(amt > bal) {
        System.out.println("Overdrawn on your checking!");
        return;
    }

    //otherwise good check above and set the new balance
    //to the current balance minus the withdrawal amount
    //via the inherited setBalance() method
    setBalance(bal-amt);
}
```

This method performs some error checking. When withdrawing from an account, the maximum you can take out is whatever your current balance is. If the amount you request to take out is more than the current balance, you will see a message print to the screen, and we return out of the method. Otherwise, proceed with the withdrawal by updating the balance of the Checking account.

Similarly with a deposit to the Checking account:

```
//method to deposit funds to the Checking account
public void deposit(double amt) {
    //get the current balance via inherited
    //getBalance() method
    double bal = getBalance();

    //if the user didn't deposit any amount, print message
    //and exit the method
    if(amt <= 0.0) {
        System.out.println("No amount requested!");
        return;
    }

    //otherwise good check above and set the new balance
    //to the current balance plus the deposit amount
```

```

        //via the inherited setBalance() method
        setBalance(bal+amt);
    }
} //Checking

```

The only error checking here is if the account holder didn't put in a deposit amount of at least a penny. Print a message and return out of the method. Otherwise, update the balance.

We will come back to the methods for transferring from one account to the other after we build the Savings account.

Savings Class

The methods for this class are basically the same; here they only affect the Savings account balances. Below is the implementation of the Savings class:

```

public class Savings extends Account{
    //one argument constructor
    public Savings(double amt) {
        super(amt);
    }

    //overloaded toString() method which prints current balance
    public String toString() {
        return "Current balance in your " +
            this.getClass().getName() + " account is: $" +
            getBalance();
    }

    //method to withdraw funds from the Savings account
    public void withdraw(double amt) {
        //get the current balance via inherited
        //getBalance() method
        double bal = getBalance();

        //if the current balance minus the requested
        //withdrawal amount will produce a negative value
        //print message and exit method
        if(bal-amt < 0.0) {
            System.out.println("Overdrawn on your savings!");
            return;
        }

        //otherwise good check above and set the new balance
        setBalance(bal-amt);
    }
}

```

```

        //to the current balance minus the withdrawal amount
        //via the inherited setBalance() method
        setBalance(bal-amt);
    }

    //method to deposit funds to the Savings account
    public void deposit(double amt) {
        //get the current balance via inherited
        //getBalance() method
        double bal = getBalance();

        //if the user didn't deposit any amount, print message
        //and exit the method
        if(amt <= 0.0) {
            System.out.println("No amount requested!");
            return;
        }

        //otherwise good check above and set the new balance
        //to the current balance plus the deposit amount
        //via the inherited setBalance() method
        setBalance(bal+amt);
    }
} //Savings

```

Test Class

For the purpose of testing, let's see a small Test class, which performs some tasks:

```

public class TestAccounts {
    public static void main(String args[]){
        //declare new bank accounts
        Savings s = new Savings(100);
        Checking c = new Checking(500);

        //check both account balances
        System.out.println("Starting balances:");
        System.out.println("Savings: $" + s.getBalance());
        System.out.println("Checking: $" + c.getBalance());

        //make a savings withdrawl
        s.withdraw(30);

        //make a checking deposit
        c.deposit(100);

        //check both account balances
    }
}

```

```

        System.out.println(s);
        System.out.println(c);

        //try a large withdrawl
        s.withdraw(1000);

        //try a large withdrawl
        c.withdraw(1000);

        //check both account balances
        System.out.println(s);
        System.out.println(c);
    } //main
} //class

```

The output when running the above program is:

Starting balances:

Savings: \$100.0

Checking: \$500.0

Current balance in your Savings account is: \$70.0

Current balance in your Checking account is: \$600.0

Overdrawn on your savings!

Overdrawn on your checking!

Current balance in your Savings account is: \$70.0

Current balance in your Checking account is: \$600.0

Transferring Funds (Checking Class)

Now we want to begin the process of transferring funds from one account to the other (Savings to Checking and Checking to Savings). Let's revisit the Checking class first by adding the method for transferring funds:

```

public class Checking extends Account{
    //one argument constructor
    public Checking(double amt) {
        super(amt);
    }

    //overloaded toString() method which prints current balance
    public String toString() {
        return "Current balance in your " +
            this.getClass().getName() + " account is: $" +

```

```

        getBalance();
    }

    //method to withdraw funds from the Checking account
    public void withdraw(double amt) {
        //get the current balance via inherited
        //getBalance() method
        double bal = getBalance();

        //if the current balance minus the requested
        //withdrawl amount will produce a negative value
        //print message and exit method
        if(amt > bal) {
            System.out.println("Overdrawn on your checking!");
            return;
        }

        //otherwise good check above and set the new balance
        //to the current balance minus the withdrawl amount
        //via the inherited setBalance() method
        setBalance(bal-amt);
    }

    //method to deposit funds to the Checking account
    public void deposit(double amt) {
        //get the current balance via inherited
        //getBalance() method
        double bal = getBalance();

        //if the user didn't deposit any amount, print message
        //and exit the method
        if(amt <= 0.0) {
            System.out.println("No amount requested!");
            return;
        }

        //otherwise good check above and set the new balance
        //to the current balance plus the deposit amount
        //via the inherited setBalance() method
        setBalance(bal+amt);
    }

    //method to transfer funds to the Checking account
    //from the Savings account
    public void transfer(Savings s, double amt) {
        //get current balances
        double checking_balance = this.getBalance();
        double savings_balance = s.getBalance();

        //if the amount requested for transfer is larger
        //then the balance from the initiating account

```

```

        //error exists so return
        if(amt > savings_balance) {
            System.out.println("Too much money for a S->C transfer!");
            return;
        }

        //good balance transfer request so update
        checking_balance += amt;
        savings_balance -= amt;

        //set respective amounts
        this.setBalance(checking_balance);
        s.setBalance(savings_balance);
    }
} //Checking

```

Here, the first argument of the *transfer()* method is of the *Savings* type, since we are looking to transfer FROM that account into the *Checking*. The method first gets the balances of both accounts. It needs to then perform an error check on the second argument, which is the amount you are requesting to transfer from one account to the other. If that amount is larger than the balance of the savings account, print a message and return out of the method. After all, much like withdrawing from one account, if the requested amount is larger than the current balance, the transaction will not be permitted. Otherwise, simply update the balances to end the method.

Transferring Funds (Savings Class)

Now let's see the same *transfer()* method in the *Savings* class:

```

public class Savings extends Account{
    //one argument constructor
    public Savings(double amt) {
        super(amt);
    }

    //overloaded toString() method which prints current balance
    public String toString() {
        return "Current balance in your " +
               this.getClass().getName() + " account is: $" +
               getBalance();
    }

    //method to withdraw funds from the Savings account
    public void withdraw(double amt) {
        //get the current balance via inherited

```

```

//getBalance() method
double bal = getBalance();

//if the current balance minus the requested
//withdrawl amount will produce a negative value
//print message and exit method
if(bal-amt < 0.0) {
    System.out.println("Overdrawn on your savings!");
    return;
}

//otherwise good check above and set the new balance
//to the current balance minus the withdrawl amount
//via the inherited setBalance() method
setBalance(bal-amt);
}

//method to deposit funds to the Savings account
public void deposit(double amt) {
    //get the current balance via inherited
    //getBalance() method
    double bal = getBalance();

    //if the user didn't deposit any amount, print message
    //and exit the method
    if(amt <= 0.0) {
        System.out.println("No amount requested!");
        return;
    }

    //otherwise good check above and set the new balance
    //to the current balance plus the deposit amount
    //via the inherited setBalance() method
    setBalance(bal+amt);
}

public void transfer(Checking c, double amt) {
    //get current balances
    double checking_balance = c.getBalance();
    double savings_balance = this.getBalance();

    //if the amount requested for transfer is larger
    //then the balance from the initiating account
    //error exists so return
    if(amt > checking_balance) {
        System.out.println("Too much money for a C->S transfer!");
        return;
    }

    //good balance transfer request so update
    savings_balance += amt;
    checking_balance -= amt;
}

```

```

        //set respective amounts
        this.setBalance(savings_balance);
        c.setBalance(checking_balance);
    }
} //Savings

```

Complete Test Class

Now let's rerun the Testing class, with some method calls for transfers:

```

public class TestAccounts {
    public static void main(String args[]){
        //declare new bank accounts
        Savings s = new Savings(100);
        Checking c = new Checking(500);

        //check both account balances
        System.out.println("Starting balances:");
        System.out.println("Savings: " + s.getBalance());
        System.out.println("Checking: " + c.getBalance());

        s.withdraw(30); //make a savings withdrawl
        c.deposit(100); //make a checking deposit

        //check both account balances
        System.out.println(s);
        System.out.println(c);

        //try a large withdrawl
        s.withdraw(1000);

        //try a large withdrawl
        c.withdraw(1000);

        //check both account balances
        System.out.println(s);
        System.out.println(c);

        //transfer to the savings from the checking
        s.transfer(c, 200);

        //check both account balances
        System.out.println(s);
        System.out.println(c);

        //transfer to the savings from the checking

```

```

        s.transfer(c, 550);

        //transfer to the checking from the savings
        c.transfer(s, 340);

        //check both account balances
        System.out.println(s);
        System.out.println(c);
    } //main
} //class

```

The output when running the above program is:

Starting balances:

Savings: 100.0

Checking: 500.0

Current balance in your Savings account is: \$70.0

Current balance in your Checking account is: \$600.0

Overdrawn on your savings!

Overdrawn on your checking!

Current balance in your Savings account is: \$70.0

Current balance in your Checking account is: \$600.0

Current balance in your Savings account is: \$270.0

Current balance in your Checking account is: \$400.0

Too much money for a C->S transfer!

Too much money for a S->C transfer!

Current balance in your Savings account is: \$270.0

Current balance in your Checking account is: \$400.0

This was a long example, but a great one to show how abstract classes work!

SHAPE CLASS

This example shows an abstract class named *Shape*, with three subclasses of *Square*, *Rectangle*, and *Circle*. Observe the code carefully before trying to determine the output of the program.

```

//Shape.java
public abstract class Shape {
    abstract double area();
}

```

```

        abstract double perimeter();
    }

//Square.java
public class Square extends Shape{
    //all 4 sides are equal in a Square
    //sides need to at least be positive so make default of 0.01
    private double sides = 0.01;

    //default constructor setting the sides equal to 1.0
    public Square(){
        sides = 1.0;
    }

    //constructor to set the sides equal to the
    //argument. Checks for a negative or 0 value
    public Square(double s) {
        if(s <= 0.0) {
            return;
        }
        sides = s;
    }

    //get/set methods
    public double getSides() { return sides; }
    public void setSides(double s) { sides = s; }

    //abstract method written in Square class
    //area of a square is its side^2
    public double area() {
        return sides * sides;
    }

    //abstract method written in Square class
    //perimeter of a square is 4*sides
    public double perimeter() {
        return 4.0 * sides;
    }

    //overloaded toString() which returns the name
    //of the class
    public String toString() {
        return getClass().getName();
    }
} //Square

//Rectangle.java
public class Rectangle extends Shape{
    //sides need to at least be positive so make default of 0.01
    private double length = 0.01, width = 0.01;
}

```

```

//default constructor setting the length & width equal to 1.0
public Rectangle(){
    length = width = 1.0;
}

//constructor to set the length & width equal to the
//arguments. Checks for a negative or 0 value for either.
public Rectangle(double l, double w) {
    if(l <= 0.0 || w <= 0.0) {
        return;
    }
    length = l;
    width = w;
}

//get/set methods
public double getLength() { return length; }
public double getWidth() { return width; }

public void setLength(double len) { length = len; }
public void setWidth(double w) { width = w; }

//abstract method written in Rectangle class
//area of a rectangle is its length * width
public double area() {
    return length * width;
}

//abstract method written in Rectangle class
//perimeter of a rectangle is 2*length + 2*width
public double perimeter() {
    return 2.0*length + 2.0*width;
}

//overloaded toString() which returns the name
//of the class
public String toString() {
    return getClass().getName();
}
} //Rectangle

//Circle.java
public class Circle extends Shape{
    private double radius = 0.01;
    private final double PI = Math.PI;

    //default constructor
    public Circle(){
        radius = 1.0;
    }
}

```

```

//constructor to set the radius equal to the
//argument. Checks for a negative or 0 value
public Circle(double r) {
    if(r <= 0) {
        return;
    }
    radius = r;
}

//get/set methods
public double getRadius() { return radius; }
public void setRadius(double r) { radius = r; }

//abstract method written in Circle class
//area of a circle is PI * radius^2
public double area() {
    return PI * radius * radius;
}

//abstract method written in Circle class
//perimeter of a circle is 2*PI*radius
//also called circumference
public double perimeter() {
    return 2.0*PI*radius;
}

//overloaded toString() which returns the name
//of the class
public String toString() {
    return getClass().getName();
}
} //Circle

//TestShape.java
public class TestShape {
    public static void main(String args[]){
        Shape sh[] = new Shape[3];
        Square sq = new Square(4);
        Rectangle re = new Rectangle(5, 10);
        Circle c1 = new Circle(8);
        Shape c2 = new Circle(2);

        sh[0] = re;
        sh[1] = c1;
        sh[2] = sq;

        //initial run
        for(int i = 0; i < 3; i++) {
            System.out.println("I'm a: " + sh[i]);
            System.out.println("My area is: " + sh[i].area());
            System.out.println("My perimeter is: "

```

```

        + sh[i].perimeter());
    }

    sh[2] = c2;
    re.setLength(3);
    re.setWidth(3);
    sh[1] = new Square();

    //after changes
    for(int i = 0; i < sh.length; i++) {
        System.out.println("I'm a: " + sh[i]);
        System.out.println("My area is: " + sh[i].area());
        System.out.println("My perimeter is: "
            + sh[i].perimeter());
    }
} //main
} //class

```

When running the above program via the `TestShape` class, this would be the output:

```

I'm a: Rectangle
My area is: 50.0
My perimeter is: 30.0
I'm a: Circle
My area is: 201.06192982974676
My perimeter is: 50.26548245743669
I'm a: Square
My area is: 16.0
My perimeter is: 16.0
I'm a: Rectangle
My area is: 9.0
My perimeter is: 12.0
I'm a: Square
My area is: 1.0
My perimeter is: 4.0
I'm a: Circle
My area is: 12.566370614359172
My perimeter is: 12.566370614359172

```

As we know by the rules of inheritance, as well as abstract classes, the parent class is `Shape`, while the `Square`, `Rectangle`, and `Circle` classes are subclasses of `Shape`. Each subclass implements the `area()` and `perimeter()` method as defined the abstract class

Shape. Each shape's area and perimeter are different, per the rules of mathematics. That is one reason why we would make each of those methods abstract.

Also present in each of the subclasses is an overloaded *toString()* method. This simply outputs the name of the class, via the use of the *getClass().getName()* methods.

Getting to the main method in the *TestShape* class, let's take it one section at a time. First, we declare an array of *Shape* objects, as well as some shapes. We construct each shape and pass it appropriate parameters.

Next, we assign shapes to the array of *Shape* objects we created. Index 0 gets a *Rectangle* object; index 1 gets a *Circle* object; and index 2 gets a *Square* object.

The first for loop prints out some information for each pass through the array. Let's see the output specifically from the first run of this loop:

```
I'm a: Rectangle  
My area is: 50.0  
My perimeter is: 125.0  
I'm a: Circle  
My area is: 201.06192982974676  
My perimeter is: 50.26548245743669  
I'm a: Square  
My area is: 16.0  
My perimeter is: 16.0
```

This outputs the name, area and perimeter of the respective object. But then we make some changes to the array and objects. The second for loop prints out the following:

```
I'm a: Rectangle  
My area is: 9.0  
My perimeter is: 18.0  
I'm a: Square  
My area is: 1.0  
My perimeter is: 4.0  
I'm a: Circle  
My area is: 12.566370614359172  
My perimeter is: 12.566370614359172
```

Why? Well, the changes we made are reflected here. We updated the length and width of the Rectangle object *re* with the use of the set methods in that object. The length and width are both set to 3, which results in an area of 9, and perimeter of 18. We also created a new *Square* and assigned that to *sh[1]*. This uses the default constructor of the *Square* class, which gives all sides a value of 1. Hence, the area is 1 and the perimeter is 4. Lastly, we gave *sh[2]* the *Shape* we created, which in this case was a *Circle*, with a radius of 2. It performs the proper calculations for the area and perimeter.

INTERFACES

When dealing with an abstract class, we sometimes see what's called an *interface*. An interface is a definition of all methods however, unlike abstract methods within an abstract class, these methods defined in the interface can be used in **any other class**, regardless of being abstract or not. Most generally, here is how to define an interface:

```
interface Name {  
    //method(s) here  
}
```

The methods that are defined in the interface **DO NOT** need the *abstract* keyword with them. In order for a class to use an interface, it must use the keyword implements in its class header. Here is the definition:

```
public class Name implements interfaceName {  
    //code here  
}
```

Where in the above, *Name* is the name of the class; and *interfaceName* is the name of the interface you will be implementing.

By default, all methods in an interface contain the keywords *public abstract* before each definition, so these are not needed. You also cannot make the methods private or protected as this is not permitted in Java interfaces.

EXAMPLE 2: Food Stores with Interfaces

Companies or stores can have their own slogans, which are catchphrases that helps a customer remember that store. Say we have 3 food stores named *BestFoods*, *OldFoods* and *WholesomeFoods*. Each of those stores have their own slogans, so an abstract method can be written to return that particular store's slogan. This can then be contained in an interface, which the parent class of *FoodStore* can implement.

```
//Slogan.java
public interface Slogan {
    String slogan();
}

//FoodStore.java
public abstract class FoodStore implements Slogan{
    private String name;
    private int opened;

    public FoodStore(String n, int o){
        if(n == null)
            throw new IllegalArgumentException("No name given!");
        else if(o < 1900 || o > 2019)
            throw new IllegalArgumentException("Bad opening!");
        else{
            name = n;
            opened = o;
        }
    } //constructor

    public String toString(){
        return("Name: " + name + "\nOpened: " + opened);
    }
} //class

//BestFoods.java
public class BestFoods extends FoodStore {
    private String slogan = "The best food anywhere!";

    public BestFoods(String n, int o){
        super(n,o);
    }

    //implements the slogan() method from the interface:
    public String slogan(){
        return slogan;
    }
} //class

//OldFoods.java
public class OldFoods extends FoodStore{
```

```

private String slogan = "The oldest food store around!";

public OldFoods(String n, int o){
    super(n,o);
}

//implements the slogan() method:
public String slogan(){
    return slogan;
}

}

//WholesomeFoods.java
public class WholesomeFoods extends FoodStore {
    private String slogan = "Mmmmmm, delicious!";

    public WholesomeFoods(String n, int o){
        super(n,o);
    }
    //implements the slogan() method:
    public String slogan(){
        return slogan;
    }
}

//Example2.java
public class Example2{
    public static void main(String args[]){
        FoodStore fs[] = new FoodStore[3];

        fs[0] = new BestFoods("Best", 1956);
        fs[1] = new OldFoods("Old", 1944);
        fs[2] = new WholesomeFoods("Whole", 1987);

        for(int i = 0; i < 3; i++)
            System.out.println(fs[i] + "\nSlogan: "
                + fs[i].slogan() + "\n");
    } //main
} //class

```

Since each subclass is inheriting all methods from the super class FoodStore, it is not necessary to place the implements Slogan in each subclass. If you did that, you would get an error citing that the method from the interface is undefined and the super class must implement the interface.

The output from running the Stores class is:

Name: Best

Opened: 1956

Slogan: The best food anywhere!

Name: Old

Opened: 1944

Slogan: The oldest food store around!

Name: Whole

Opened: 1987

Slogan: Mmmmmm, delicious!

EXAMPLE 3: An Interface without an Abstract Class

Here is an example not using an abstract class:

```
//Sounds.java
public interface Sounds{
    public String sounds();
}

//Animal.java
public class Animal implements Sounds{
    private String name = "";
    Animal(String n){
        name = n;
    }

    //implemented sounds() method
    public String sounds() {
        return "makes no sounds!";
    }

    //overloaded toString() method
    public String toString() {
        return "My name is " + name
        + " and I am a " + getClass().getName()
        + " who " + sounds();
    }
}

//Dog.java
public class Dog extends Animal{
    Dog(String n){
        super(n);
    }
}
```

```

}

//implemented sounds() method
public String sounds() {
    return "barks!";
}

//overloaded toString() method
public String toString() {
    return super.toString();
}
}

//Cat.java
public class Cat extends Animal {
    Cat(String n){
        super(n);
    }

    //implemented sounds() method
    public String sounds() {
        return "meows!";
    }

    //overloaded toString() method
    public String toString() {
        return super.toString();
    }
}

//Lion.java
public class Lion extends Animal {
    Lion(String n){
        super(n);
    }

    //implemented sounds() method
    public String sounds() {
        return "roars!";
    }

    //overloaded toString() method
    public String toString() {
        return super.toString();
    }
}

//Example3.java
public class Example3{
    public static void main(String args[]){
        Dog d = new Dog("Ralph");
    }
}

```

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

Cat c = new Cat("Lucy");
System.out.println(c);

Lion l = new Lion("Max");
System.out.println(l);

Animal a = new Animal("Steve");
System.out.println(a);
} //main
} //class
```

The output from running the above Example3 program is:

```
My name is Ralph and I am a Dog who barks!
My name is Lucy and I am a Cat who meows!
My name is Max and I am a Lion who roars!
My name is Steve and I am a Animal who makes no sounds!
```

The Animal class implements the Sounds interface. By inheritance, the subclasses Dog, Cat, and Lion inherit the need to implement the sounds() method.

CHAPTER 18

Threading & Multitasking

This chapter introduces a powerful feature of Java called threading. This chapter shows how you can multitask, and run objects in the background of a larger program.

TOPICS

- | | |
|-----------------------|-----|
| 1. <i>Overview</i> | 396 |
| 2. <i>Car Example</i> | 398 |

In Java, a **thread** is a smaller piece of a larger program. It also can be a super class to objects in a program.

For example, say you wanted to simulate some cars driving, or horses racing at a track; you can do all that with threading. Using the first example above, we will expand on that in this chapter.

So what are some useful methods of the Thread super class?

void sleep(long milliseconds)

This method will delay the program for a specified amount of time. Keep in mind that 1000 milliseconds are equivalent to 1 second. So if you wanted the program to delay for 10 seconds, the argument to the method will be the value 10000. Also know that the method is accessed as such:

`Thread.sleep(...);`

And **requires** a try/catch block looking for an *InterruptedException* (for now, that kind of exception will not be thrown here with the program we will write because we are not dealing with monitors in Java, a more advanced threading concept).

void start()

This method will be called from the Thread object itself and will invoke the `run()` method within that object. This will then start the thread.

void run()

This method will **need to be** overloaded in the Thread object itself. It is a required overload and will act like the main method of an object. This is where the code will be implemented.

EXAMPLE 1: *Countdown Clock*

The below program will countdown from the number the user enters. Number must be between 5 and 30.

```
import java.util.Scanner;
public class Example1{
    public static void main(String args[]){
        Scanner s = new Scanner(System.in);

        //prompt user for entry
```

```

int entry = 0;
System.out.print("Enter a number between 5 and 30: ");
entry = s.nextInt();

//entry is invalid so keep trying
while(entry < 5 || entry > 30) {
    System.out.println("Try again: ");
    entry = s.nextInt();
}

try {
    System.out.println("Program ends in ");

    //loop through to print a number every 1 second
    for(int i = entry; i >= 1; i--) {
        System.out.print(i + "...");
        Thread.sleep(1000);
    }
} catch(Exception e) {
    System.out.println(e);
}
} //main
} //class

```

The program will print a number to the console every 1 second. A sample output from running the above program is:

```

Enter a number between 5 and 30: 3
Try again: 2
Try again: 10
Program ends in
10....9...8....7...6....5....4....3....2....1...

```

EXAMPLE 2: Print a Box of Stars

The below program will print a box of stars to the console. It prints to the console every 0.5 seconds.

```

import java.util.Scanner;
public class Example2{
    public static void main(String args[]){
        Scanner s = new Scanner(System.in);

        //prompt user for entry

```

```

int size = 0;
System.out.print("Enter a number between 3 and 10: ");
size = s.nextInt();

//out of range so try again
while(size < 3 || size > 10) {
    System.out.println("Try again: ");
    size = s.nextInt();
}

try {
    //print the box of stars
    for(int r = 1; r <= size; r++) {
        for(int c = 1; c <= size; c++) {
            //print a star or space every 1/2 second
            Thread.sleep(500);
            if(c == 1 || c == size
            || r == 1 || r == size){
                System.out.print("*");
            }else{
                System.out.print(" ");
            }
        }
        System.out.println();
    }
} catch(Exception e) {
    System.out.println(e);
}
} //main
} //class

```

A sample output from the above program is:

Enter a number between 3 and 10: 5

```

*****
*   *
*   *
*   *
*****

```

CAR EXAMPLE

For now, the skeleton of the example will deal with a class called *Car*. It will extend the super class *Thread*, with contains all the necessary methods for simulation.

```
public class Car extends Thread{
    //code ...
}
```

In addition, we want to know the model of the car we are dealing with for the purpose of denoting which thread is printing a message.

Here is a basic *Car* class with explanations following each block of code.

```
//Car.java
import java.util.Random;
public class Car extends Thread{
    private String model;

    public Car(String m){
        model = m;
    }
    //overloaded run() method
    public void run(){
        Random r = new Random();
        try{
            //from 1 to 6 seconds
            Thread.sleep(r.nextInt(5000)+1000);
            startCar();

            //from 1 to 4 seconds
            Thread.sleep(r.nextInt(3000)+1000);
            drive();

            //from 3 to 13 seconds
            Thread.sleep(r.nextInt(10000)+3000);
            arriveDest();

            //from 1 to 4 seconds
            Thread.sleep(r.nextInt(3000)+1000);
            drive();

            //from 3 to 13 seconds
            Thread.sleep(r.nextInt(10000)+3000);
            arriveHome();

        }catch(InterruptedException ie){
            System.out.println(model + " interrupted...");
        }
    }
}
```

```

private void startCar(){
    System.out.println(model + " started");
}
private void drive(){
    System.out.println(model + " is running!");
}
private void arriveDest(){
    System.out.println(model + " arrived at destination!");
}
private void arriveHome(){
    System.out.println(model + " arrived back home!");
}
} //class Car

```

This class will be the Thread class for the program. We need to point out a few things from the above.

Notice that each time the *sleep()* method is called, it is called from *Thread.sleep()*. This is **not** an inherited method from the super class Thread. Also, the *run()* method is overloaded as that is a requirement.

The other methods of the class are methods used to print messages to let us know what is going on with the simulation. Everything will be randomized and will be different each time the program is run. But where is our main class to start the simulation? Here is it below:

```

public class MainCars{
    public static void main(String args[]){
        Car cars[] = new Car[5];

        cars[0] = new Car("Ford Taurus");
        cars[1] = new Car("Mazda Tribute");
        cars[2] = new Car("Toyota Camary");
        cars[3] = new Car("Honda Accord");
        cars[4] = new Car("Ford Tundra");

        for(int i = 0; i < 5; i++){
            cars[i].start();
        }
    } //main
} //class

```

This class will be used to “start the cars” for the simulation. So what output will be produced? As mentioned, it will be different each time the program is run, so here are three different outputs to the program:

TRIAL 1:

Mazda Tribute started
Honda Accord started
Honda Accord is running!
Ford Tundra started
Mazda Tribute is running!
Toyota Camary started
Ford Taurus started
Ford Tundra is running!
Toyota Camary is running!
Ford Taurus is running!
Honda Accord arrived at destination!
Mazda Tribute arrived at destination!
Mazda Tribute is heading home!
Honda Accord is heading home!
Ford Tundra arrived at destination!
Ford Taurus arrived at destination!
Toyota Camary arrived at destination!
Ford Taurus is heading home!
Ford Tundra is heading home!
Toyota Camary is heading home!
Mazda Tribute arrived back home!
Ford Taurus arrived back home!
Honda Accord arrived back home!
Ford Tundra arrived back home!
Toyota Camary arrived back home!

TRIAL 2:

Mazda Tribute started
Ford Tundra started
Honda Accord started
Ford Taurus started
Toyota Camary started
Mazda Tribute is running!
Ford Tundra is running!

Ford Taurus is running!
Honda Accord is running!
Toyota Camary is running!
Honda Accord arrived at destination!
Ford Taurus arrived at destination!
Mazda Tribute arrived at destination!
Honda Accord is heading home!
Ford Taurus is heading home!
Mazda Tribute is heading home!
Ford Tundra arrived at destination!
Toyota Camary arrived at destination!
Honda Accord arrived back home!
Ford Tundra is heading home!
Toyota Camary is heading home!
Mazda Tribute arrived back home!
Ford Taurus arrived back home!
Toyota Camary arrived back home!
Ford Tundra arrived back home!

TRIAL 3:

Ford Tundra started
Mazda Tribute started
Honda Accord started
Toyota Camary started
Ford Taurus started
Ford Tundra is running!
Toyota Camary is running!
Mazda Tribute is running!
Ford Taurus is running!
Honda Accord is running!
Ford Tundra arrived at destination!
Toyota Camary arrived at destination!
Honda Accord arrived at destination!
Ford Tundra is heading home!
Mazda Tribute arrived at destination!
Toyota Camary is heading home!
Honda Accord is heading home!
Mazda Tribute is heading home!
Ford Taurus arrived at destination!

```
Ford Tundra arrived back home!
Ford Taurus is heading home!
Honda Accord arrived back home!
Toyota Camary arrived back home!
Mazda Tribute arrived back home!
Ford Taurus arrived back home!
```

Take careful note that each output is different, since it is dealing with random numbers.

Now that we have the basics down, let's add another feature to the class. Most of the time, people have to run multiple errands (getting gas, doing laundry, grocery shopping, etc...) We can create another constructor of the class, taking an additional argument that represents the number of errands each car will run. If it is just one, we can make use of the original one argument constructor.

Here are the full modified classes to the above example. Output will follow similar to the above.

```
//Car.java
import java.util.Random;
public class Car extends Thread{
    private String model;
    private int chores = 1; //default value

    public Car(String m){
        model = m;
    }

    public Car(String m, int c){
        model = m;
        chores = c;
    }

    public void run(){
        Random r = new Random();

        try{
            //sleep 1 to 6 seconds
            Thread.sleep(r.nextInt(5000)+1000);
            startCar();

            //loop through based on the number of chores the
            //driver has to make
            for(int i = 0; i < chores; i++){
                //sleep 1 to 4 seconds
                Thread.sleep(r.nextInt(3000)+1000);
                startCar();
            }
        } catch(InterruptedException e) {
            e.printStackTrace();
        }
    }

    void startCar(){
        System.out.println(model + " started!");
    }
}
```

```

        Thread.sleep(r.nextInt(3000)+1000);
        drive(false); //more chores to do

        //sleep 3 to 13 seconds
        Thread.sleep(r.nextInt(10000)+3000);
        arriveDest(i+1);
    }

    //sleep 1 to 4 seconds
    Thread.sleep(r.nextInt(3000)+1000);
    drive(true); //no more chores

    //sleep 3 to 13 seconds
    Thread.sleep(r.nextInt(10000)+3000);
    arriveHome();
} catch(InterruptedException ie){
    System.out.println(model + " interrupted...");
}
}

private void startCar(){
    System.out.println(model + " started");
}

//modified method from before
private void drive(boolean home){
    if(home){
        System.out.println(model + " is heading home!");
    } else{
        System.out.println(model + " is running!");
    }
}

private void arriveDest(int n){
    System.out.println(model + " arrived at destination " + n +
        " of " + chores);
}

private void arriveHome(){
    System.out.println(model + " arrived back home!");
}
}

//MainCars.java
public class MainCars{
    public static void main(String args[]){
        Car cars[] = new Car[5];

        cars[0] = new Car("Ford Taurus", 3);
        cars[1] = new Car("Mazda Tribute", 2);
        cars[2] = new Car("Toyota Camary");
        cars[3] = new Car("Honda Accord", 2);
        cars[4] = new Car("Ford Tundra");
    }
}

```

```
for(int i = 0; i < 5; i++){
    cars[i].start();
}
} //main
} //class
```

The three output trials are as follows.

TRIAL 1:

Toyota Camary started
Mazda Tribute started
Ford Tundra started
Toyota Camary is running!
Mazda Tribute is running!
Ford Taurus started
Ford Tundra is running!
Honda Accord started
Honda Accord is running!
Ford Taurus is running!
Ford Tundra arrived at destination 1 of 1
Honda Accord arrived at destination 1 of 2
Toyota Camary arrived at destination 1 of 1
Ford Tundra is heading home!
Honda Accord is running!
Toyota Camary is heading home!
Mazda Tribute arrived at destination 1 of 2
Ford Taurus arrived at destination 1 of 3
Mazda Tribute is running!
Ford Taurus is running!
Honda Accord arrived at destination 2 of 2
Ford Tundra arrived back home!
Toyota Camary arrived back home!
Honda Accord is heading home!
Ford Taurus arrived at destination 2 of 3
Ford Taurus is running!
Mazda Tribute arrived at destination 2 of 2
Honda Accord arrived back home!
Mazda Tribute is heading home!
Ford Taurus arrived at destination 3 of 3

Ford Taurus is heading home!
Mazda Tribute arrived back home!
Ford Taurus arrived back home!

TRIAL 2:

Ford Tundra started
Ford Tundra is running!
Mazda Tribute started
Toyota Camary started
Honda Accord started
Mazda Tribute is running!
Ford Taurus started
Toyota Camary is running!
Honda Accord is running!
Ford Taurus is running!
Ford Tundra arrived at destination 1 of 1
Honda Accord arrived at destination 1 of 2
Ford Tundra is heading home!
Honda Accord is running!
Mazda Tribute arrived at destination 1 of 2
Ford Taurus arrived at destination 1 of 3
Toyota Camary arrived at destination 1 of 1
Ford Tundra arrived back home!
Mazda Tribute is running!
Ford Taurus is running!
Toyota Camary is heading home!
Mazda Tribute arrived at destination 2 of 2
Ford Taurus arrived at destination 2 of 3
Mazda Tribute is heading home!
Honda Accord arrived at destination 2 of 2
Ford Taurus is running!
Toyota Camary arrived back home!
Honda Accord is heading home!
Mazda Tribute arrived back home!
Ford Taurus arrived at destination 3 of 3
Honda Accord arrived back home!
Ford Taurus is heading home!
Ford Taurus arrived back home!

TRIAL 3:

Ford Taurus started
Honda Accord started
Ford Tundra started
Honda Accord is running!
Ford Tundra is running!
Ford Taurus is running!
Mazda Tribute started
Toyota Camary started
Mazda Tribute is running!
Ford Tundra arrived at destination 1 of 1
Toyota Camary is running!
Honda Accord arrived at destination 1 of 2
Ford Tundra is heading home!
Honda Accord is running!
Ford Taurus arrived at destination 1 of 3
Ford Tundra arrived back home!
Ford Taurus is running!
Mazda Tribute arrived at destination 1 of 2
Toyota Camary arrived at destination 1 of 1
Honda Accord arrived at destination 2 of 2
Toyota Camary is heading home!
Mazda Tribute is running!
Honda Accord is heading home!
Toyota Camary arrived back home!
Honda Accord arrived back home!
Mazda Tribute arrived at destination 2 of 2
Ford Taurus arrived at destination 2 of 3
Mazda Tribute is heading home!
Ford Taurus is running!
Mazda Tribute arrived back home!
Ford Taurus arrived at destination 3 of 3
Ford Taurus is heading home!
Ford Taurus arrived back home!

As you can see, this chapter was just a basic understanding of how threads work. Feel free to expand on this program as much as you like. It is only to your benefit to do so.

CHAPTER 19

Introduction to Generics, Collections & Enums

This chapter will give an overview of Generics in Java. This can be thought of as writing methods and objects that can handle multiple data types. It also discusses what an Enum is.

TOPICS

1. <i>Generic Methods</i>	409
2. <i>Generic Classes</i>	415
3. <i>Enums</i>	420

In our Java adventures thus far, we have learned about methods and how they are utilized. We wrote methods to perform arithmetic operations, to find the largest or smallest values, or even to determine if an element is contained in an array or object. Each of those methods uses a certain data type, whether its primitive or object.

Let's say that we wanted to find the smallest element in an array, and write a method to do so. We would have to write a method for 5 different data types (int, short, long, float and double) as each are specific. What if I told you that we can write one method to handle that task, and not worry about data types? It's possible with the help of what's called a **generic method**.

GENERIC METHODS

There are some rules about declaring a generic method:

1. All generic method declarations have a **type parameter** that is delimited by **angle brackets** `< >`, and that precedes the methods return type.
2. The methods' parameters and the method return type can be of said type parameter.
3. A generic method's body is declared and utilized like that of any other method.
4. The type parameters for a generic method can only be of the wrapper class type or custom object, NOT a primitive data type.

So most generally, a generic method is declared as such:

```
[identifier] < T > [return type] method_name ( arguments );
```

Where in the above, `[identifier]` can be either *public*, *private*, *protected* or nothing; `< T >` is the type parameter being used for the method; `return_type` is the return type of the method (can be void); `method_name` is a useful name for the method; and `arguments` are any parameters you need to pass for the method.

Some examples will clear this up.

EXAMPLE 1: Printing Elements in an Array

```
public class Example1{
    private static <T> void printElements(T[] arr) {
```

```

        for(T val : arr) {
            System.out.print(val + " ");
        }
        System.out.println();
    }

    public static void main(String args[]){
        Integer arr1[] = {1, 2, 3, 4, 5};
        Double arr2[] = {1.0, 2.0, 3.0, 4.0, 5.0};
        Character arr3[] = {'A', 'B', 'C', 'D', 'E'};

        System.out.println("Printing arr1");
        printElements(arr1);

        System.out.println("Printing arr2");
        printElements(arr2);

        System.out.println("Printing arr3");
        printElements(arr3);
    } //main
} //class

```

The generic method will print elements from any type array. It utilizes a foreach loop to perform the printing.

The output from running the above program is:

```

Printing arr1
1 2 3 4 5
Printing arr2
1.0 2.0 3.0 4.0 5.0
Printing arr3
A B C D E

```

Bounded Parameter Types

In the above example, this shows a potential flaw in the process of writing generics. Let's say that we wanted to write a generic method to handle finding the maximum value in an array. Generally, that deals with numbers (Integer, Double, etc). But because of the generics of the method, we can legally pass it a String or custom object type. This will not produce exactly what we want.

That problem can be solved by **bounding** our type parameters. When declaring the type parameter of a method, we will use the keyword **extends**.

Let's write the header line for a generic method to find the largest value of two parameters of type T.

```
public <T extends Number> T something(T a, T b);
```

This will bound the method to any subclass of Number, which in this case is Integer, Short, Long, Double, and Float. If we attempted to pass a String to the method, it will result in a compiler error.

The below example will demonstrate this.

EXAMPLE 2: Finding Maximum and Minimum Values

This program will find the maximum and minimum values when comparing three elements.

```
public class Example2{
    private static <T extends Comparable<T>> T max(T a, T b, T c) {
        //start at first element
        T max_value = a;

        //if next element comes after current max_value
        //then new max_value is b
        if(b.compareTo(max_value) > 0)
            max_value = b;

        //if next element comes after current max_value
        //then new max_value is c
        if(c.compareTo(max_value) > 0)
            max_value = c;

        return max_value;
    }
    private static <T extends Comparable<T>> T min(T a, T b, T c) {
        //start at first element
        T min_value = a;

        //if next element comes before current min_value
        //then new min_value is b
```

```

        if(b.compareTo(min_value) < 0)
            min_value = b;

        //if next element comes before current min_value
        //then new min_value is c
        if(c.compareTo(min_value) < 0)
            min_value = c;

        return min_value;
    }
    public static void main(String args[]){
        Integer a = 5, b = 3, c = 7;
        Double i = 4.44, j = 4.32, k = 4.44;
        Character x = 'C', y = 'R', z = 'K';

        System.out.println( max(a, b, c) );
        System.out.println( max(i, j, k) );
        System.out.println( max(x, y, z) );

        System.out.println( min(a, b, c) );
        System.out.println( min(i, j, k) );
        System.out.println( min(x, y, z) );
    } //main
} //class

```

The output from running the above program is:

```

5
4.44
R
3
4.32
C

```

EXAMPLE 3: Finding Maximum and Minimum Values in an Array

This program will find the maximum and minimum values in arrays of different types.

```

public class Example3{
    private static <T extends Comparable<T>> T max(T[] arr) {
        //start at first element
        T max_value = arr[0];

        for(T curr : arr) {

```

```

        //first comparison will return 0
        if(curr.compareTo(max_value) > 0)
            max_value = curr;
    }
    return max_value;
}
private static <T extends Comparable<T>> T min(T[] arr) {
    //start at first element
    T min_value = arr[0];

    for(T curr : arr) {
        //first comparison will return 0
        if(curr.compareTo(min_value) < 0)
            min_value = curr;
    }
    return min_value;
}

public static void main(String args[]){
    Short arr[] = {5, 12, 4, 5, 2, 7};
    Integer arr2[] = {15, 12, 24, 5, 12, 37};
    Double arr3[] = {5.5, 12.2, 1.7, 12.4, 12.3, 12.1};
    String arr4[] = {"Frank","Ralph","Lisa","Charles","James"};

    System.out.println( max(arr) );
    System.out.println( max(arr2) );
    System.out.println( max(arr3) );
    System.out.println( max(arr4) );
    System.out.println( max(arr) );
    System.out.println( min(arr2) );
    System.out.println( min(arr3) );
    System.out.println( min(arr4) );
} //main
} //class

```

The output from running the above program is:

```

12
37
12.4
Ralph
12
5
1.7
Charles

```

Multiple Parameter Types

Any generic method, or class, can have multiple generic types associated with it. Each type is separated by a comma in the method declaration.

```
[identifier] < A, B, C ... > [return type] method_name ( arguments );
```

However, there can only be one of those defined as the method return type.

EXAMPLE 4: Multiple Parameter Types

The below program will print elements of an array of types A and B.

```
public class Example4{
    private static <A, B> void mystery(A a[], B b[]) {
        int sizeA = a.length, sizeB = b.length;

        for(int i = sizeA-1; i >= 0; i--) {
            System.out.print(a[i]);
        }
        System.out.println();

        for(int j = 0; j < sizeB; j++) {
            System.out.print(b[j]);
        }
        System.out.println();
    }

    public static void main(String args[]){
        Character arr[] = {'o', 'l', 'l', 'e', 'H'};
        Integer arr2[] = {9, 0, 2, 1, 0};

        mystery(arr, arr2);
    } //main
} //class
```

The output from running the above program is:

Hello
90210

EXAMPLE 5: *Multiple Bounded Parameter Types*

The below program modifies the above example, bounding the B type parameter to a numerical type.

```
public class Example5{
    private static <A, B extends Number> void mystery(A a[], B b[]) {
        int sizeA = a.length, sizeB = b.length;

        for(int i = sizeA-1; i >= 0; i--) {
            System.out.print(a[i]);
        }
        System.out.println();

        for(int j = 0; j < sizeB; j++) {
            System.out.print(b[j]);
        }
        System.out.println();
    }

    public static void main(String args[]){
        Character arr[] = {'o', 'l', 'l', 'e', 'H'};
        Integer arr2[] = {9, 0, 2, 1, 0};

        mystery(arr, arr2);
    } //main
} //class
```

The output is the same as Example 4.

GENERIC CLASSES

Just like creating a generic method, we can also create a **generic class**. This follows the same declarations of any class, with exception of the type parameter being declared.

Most generally, a generic class is defined as such:

```
public class Gen <T> {
    //code
}
```

EXAMPLE 6: *Most Generic Class*

Below shows the most generic of classes that we can define, and shows some examples of its use.

```
//Gen.java
public class Gen<T>{
    T data;

    Gen(T data) {
        this.data = data;
    }
    public T getData() { return data; }
    public void setData(T data) {
        this.data = data;
    }
} //Gen

public class Example6{
    public static void main(String args[]){
        Gen<Integer> g = new Gen<Integer> (15);
        Gen<Double> g2 = new Gen<Double> (15.0);
        Gen<String> g3 = new Gen<String> ("Hello");

        //print data
        System.out.println(g.getData());
        System.out.println(g2.getData());
        System.out.println(g3.getData());
    } //main
} //class
```

The output from running the above program is:

```
15
15.0
Hello
```

Bounded Parameter Types

Using the example from the generic method section about bounding type parameters, let's say we wanted to use a generic class that handles numbers. We can declare the following:

```
public class Gen<T extends Number> {  
    //code  
}
```

This will bound the class to any numerical type (Integer, Double, etc).

The below declarations are legal and acceptable, using the above *Gen* class:

```
Gen<Integer> num = new Gen<Integer>();  
Gen<Integer> num2 = new Gen<Integer>(45);  
Gen<Double> num3 = new Gen<Double>(4.5);  
Gen<Long> num4 = new Gen<Long>(1234567890);  
Gen<Short> num5 = new Gen<Short>();
```

Etc...

The following declarations are illegal, and will result in compiler errors using the above *Gen* class.

```
Gen<String> num = new Gen<String>();  
Gen<Boolean> num = new Gen<Boolean>(true);  
Gen<Character> num = new Gen<Character>();  
Gen<String> num = new Gen<String>("Wrong");
```

The below example bounds the generic class *Gen* to a numerical type.

```
public class Gen<T extends Number>{  
    T data;  
    Gen(T data){  
        this.data = data;  
    }  
    public T getData() { return data; }  
    public void setData(T data) {  
        this.data = data;  
    }  
}  
  
public class Example{  
    public static void main(String args[]){  
        Gen<Integer> g = new Gen<Integer> (15);  
    }  
}
```

```

        Gen<Double> g2 = new Gen<Double> (15.0);

        System.out.println(g.getData());
        System.out.println(g2.getData());

    } //main
} //class

```

Multiple Parameter Types

A generic class can have multiple, generic types associated with it. Each type is separated by a comma in the class declaration.

Most generally, a multi-type generic class is defined as such:

```

public class Gen <T1, T2 ...> {
    //code
}

```

EXAMPLE 7: Multi-type Generic Class

This example will show a class called *NameAge*, that will take two data types, String and Integer, representing data for a user's name and age. Values are entered from the console.

```

//NameAge.java
public class NameAge<A, B extends Number>{
    A name;
    B age;

    public NameAge(A a, B b) {
        name = a;
        age = b;
    }

    public String toString() {
        return this.name + ", " + this.age + " years old";
    }
} //NameAge

//Example7.java

```

```
import java.util.Scanner;
public class Example7{
    private static void print(NameAge[] arr) {
        for(NameAge curr : arr) {
            System.out.println(curr);
        }
    }

    public static void main(String args[]){
        Scanner s = new Scanner(System.in);

        NameAge<String, Integer> na[] = new NameAge[3];

        String name;
        Integer age;

        for(int i = 0; i < 3; i++) {
            System.out.println("Please enter your name: ");
            name = s.next();

            System.out.println("Please enter your age: ");
            age = s.nextInt();

            na[i] = new NameAge<String, Integer> (name, age);
        }
        print(na);
    } //main
} //class
```

A sample output from running the above program is:

Please enter your name:

Alex

Please enter your age:

32

Please enter your name:

Lisa

Please enter your age:

28

Please enter your name:

Frank

Please enter your age:

44

ENUMS

An **enum** is a special data type that can be thought of as a collection of constants. It can be quite useful when dealing with items that you know are not going to change, such as compass directions, days of the week, months of the year, playing cards, etc.

Most generally, an enum is defined as such:

```
public enum Name{  
    //values  
}
```

Where in the above, *Name* is a useful name for the enum object.

So let's see an example of compass directions:

```
public enum Compass{  
    NORTH,  
    SOUTH,  
    EAST,  
    WEST  
}
```

When dealing with enums, we need to keep these things in mind:

1. The enum is also a data type (in the above example, *Compass*).
2. Each enum value is separated by a comma.
3. Each enum value has **public static final** before it (this is done automatically in Java and not needed to be written).
4. Since these are constant values, a good practice is to capitalize the names.

But how do we assign a value to an enum type? An example is below.

```
Compass direction = Compass.SOUTH;
```

Let's see a more in depth example using the *Compass* enum.

EXAMPLE 8: *Compass Directions*

```
//Compass.java
public enum Compass{
    NORTH,
    SOUTH,
    EAST,
    WEST
}

//Example8.java
public class Example8{
    public static void main(String args[]){
        Compass c = Compass.WEST;

        if(c == Compass.NORTH){
            System.out.println("Heading North");
        }else if(c == Compass.SOUTH){
            System.out.println("Heading South");
        }else if(c == Compass.EAST){
            System.out.println("Heading East");
        }else{
            System.out.println("Heading West");
        }

        //change direction
        c = Compass.SOUTH;

        switch(c){
            case NORTH:
                System.out.println("Now Heading North");
                break;
            case SOUTH:
                System.out.println("Now Heading South");
                break;
            case EAST:
                System.out.println("Now Heading East");
                break;
            case WEST:
                System.out.println("Now Heading West");
                break;
        }
    } //main
} //class
```

The output when running the above program is:

Heading West
Now Heading South

Iterating an Enum

Say you want to display each enum value. We can iterate through the enum with the use of a foreach loop. This is the easiest way to navigate an enum.

Using the above Compass enum again, let's display each value:

```
for(Compass c : Compass.values()){
    System.out.println(c);
}
```

The *values()* method will return all values in the enum.

Enum Methods and Fields

An enum, while containing constant values, can also contain other elements of a class, such as variables, methods, and constructors. Let's expand on the Compass example above.

```
public enum Compass{
    NORTH ("N"),
    SOUTH ("S"),
    EAST ("E"),
    WEST ("W");

    //instance variable
    private final String code;

    //constructor
    Compass(String code){ this.code = code; }

    //get method
    public String getDirection(){ return this.code; }
}
```

Each enum value will now call the constructor and pass to it the value contained in parentheses (N, S, E, W, respectively). It is very important to note that once we are done

declaring each enum value, we must include a semicolon, otherwise there would be a compiler error. Also contained in this enum is an instance variable named *code*, that will hold its value, as well as a get method named *getDirection()*, that will return its value. Notice there is no set method needed as these values are constants.

Let's expand on this further in the below example:

EXAMPLE 9: *Compass Directions II*

```
//Compass.java
public enum Compass{
    NORTH ("N"),
    SOUTH ("S"),
    EAST ("E"),
    WEST ("W");

    //instance variable
    private final String code;

    //constructor
    Compass(String code){ this.code = code; }

    //get method
    public String getDirection(){ return this.code; }
}

//Example9.java
public class Example9{
    public static void main(String args[]){
        Compass c = Compass.WEST;

        System.out.println("We are heading " + c.getDirection());

        //change direction
        c = Compass.SOUTH;

        System.out.println("We are now heading "
            + c.getDirection());
    } //main
} //class
```

The output from running the above program is:

We are heading W

We are now heading S

Abstract Methods & Enums

While we saw the above *Compass* example contain a simple instance variable and method, let's see another one that contains an abstract method. Here, we will be dealing with a traffic signal that contains Red, Yellow, and Green lights. Each instance of the enum will implement a *nextSignal()* method, which we define as abstract.

```
public enum TrafficLight{
    RED (60) {
        public TrafficLight nextSignal(){ return GREEN; }
    },
    YELLOW (10) {
        public TrafficLight nextSignal(){ return RED; }
    },
    GREEN (60) {
        public TrafficLight nextSignal(){ return YELLOW; }
    };

    //abstract method
    public abstract TrafficLight nextSignal();

    //instance variable
    private final int secondsAtColor;

    //constructor
    TrafficLight(int secondsAtColor){
        this.secondsAtColor = secondsAtColor;
    }

    //get method
    public int getSecondsAtColor(){ return this.secondsAtColor; }
}
```

Each enum value will implement the abstract method *nextSignal()*. The return type is of *TrafficLight*, as we want to know what color light will be triggered next. As we know, once a light is red, the next color is green; once it is green, the next color is yellow; and finally, red. The numeric values indicate the number of seconds each color is present on the traffic signal. Let's expand on this with an example using the above enum.

EXAMPLE 10: Traffic Signals

```

//TrafficLight.java
public enum TrafficLight{
    RED (60){
        public TrafficLight nextSignal(){ return GREEN; }
    },
    YELLOW (10) {
        public TrafficLight nextSignal(){ return RED; }
    },
    GREEN (60) {
        public TrafficLight nextSignal(){ return YELLOW; }
    };

    //abstract method
    public abstract TrafficLight nextSignal();

    //instance variable
    private final int secondsAtColor;

    //constructor
    TrafficLight(int secondsAtColor){
        this.secondsAtColor = secondsAtColor;
    }

    //get method
    public int getSecondsAtColor(){ return this.secondsAtColor; }
}

//Example10.java
public class Example10{
    public static void main(String args[]){
        for(TrafficLight tl : TrafficLight.values()){
            System.out.println("Light is currently " + tl);
            System.out.println("It will last for: "
                + tl.getSecondsAtColor());
            System.out.println("Next color is: "
                + tl.nextSignal());
        }
    } //main
} //class

```

The output from running the above program is:

Light is currently RED
 It will last for: 60
 Next color is: GREEN
 Light is currently YELLOW

It will last for: 10
Next color is: RED
Light is currently GREEN
It will last for: 60
Next color is: YELLOW

EXAMPLE 11: *Days of the Week*

This example will show an enum of days of the week (Sunday – Saturday). It will also implement two abstract methods, which will return the value of its previous day, and its next day.

```
//DayOfTheWeek.java
public enum DayOfTheWeek{
    MONDAY (1){
        public DayOfTheWeek previous(){ return SUNDAY; }
        public DayOfTheWeek next(){ return TUESDAY; }
    },
    TUESDAY (2){
        public DayOfTheWeek previous(){ return MONDAY; }
        public DayOfTheWeek next(){ return WEDNESDAY; }
    },
    WEDNESDAY (3){
        public DayOfTheWeek previous(){ return TUESDAY; }
        public DayOfTheWeek next(){ return THURSDAY; }
    },
    THURSDAY (4){
        public DayOfTheWeek previous(){ return WEDNESDAY; }
        public DayOfTheWeek next(){ return FRIDAY; }
    },
    FRIDAY (5){
        public DayOfTheWeek previous(){ return THURSDAY; }
        public DayOfTheWeek next(){ return SATURDAY; }
    },
    SATURDAY (6){
        public DayOfTheWeek previous(){ return FRIDAY; }
        public DayOfTheWeek next(){ return SUNDAY; }
    },
    SUNDAY (7){
        public DayOfTheWeek previous(){ return SATURDAY; }
        public DayOfTheWeek next(){ return MONDAY; }
    };

    //abstract methods
    public abstract DayOfTheWeek previous();
    public abstract DayOfTheWeek next();
}
```

```

//instance variable
private final int value;

//constructor
DayOfTheWeek(int value){
    this.value = value;
}

//get methods
public int getValue(){ return this.value; }
}

//Example11.java
import java.util.Scanner;
public class Example11{
    public static void main(String args[]){
        Scanner s = new Scanner(System.in);
        System.out.println("Enter a number between 1 & 7: ");
        int val = s.nextInt();

        while(val < 1 || val > 7){
            System.out.println("Try again: ");
            val = s.nextInt();
        }

        for(DayOfTheWeek day : DayOfTheWeek.values()){
            if(day.getValue() == val){
                System.out.println("You chose " + day);
                System.out.println("It is preceded by " +
                    + day.previous());
                System.out.println("It is followed by " +
                    + day.next());
                break;
            }
        }
    } //main
} //class

```

A sample out from the above program is:

Enter a number between 1 & 7:

4

You chose THURSDAY

It is preceded by WEDNESDAY

It is followed by FRIDAY

Enums & Arrays

We can also make use of the `values()` method to break the enum into an array of that type. The above example is modified to highlight this.

EXAMPLE 12: Days of the Week II

```
//DayOfTheWeek.java
public enum DayOfTheWeek{
    MONDAY (1){
        public DayOfTheWeek previous(){ return SUNDAY; }
        public DayOfTheWeek next(){ return TUESDAY; }
    },
    TUESDAY (2){
        public DayOfTheWeek previous(){ return MONDAY; }
        public DayOfTheWeek next(){ return WEDNESDAY; }
    },
    WEDNESDAY (3){
        public DayOfTheWeek previous(){ return TUESDAY; }
        public DayOfTheWeek next(){ return THURSDAY; }
    },
    THURSDAY (4){
        public DayOfTheWeek previous(){ return WEDNESDAY; }
        public DayOfTheWeek next(){ return FRIDAY; }
    },
    FRIDAY (5){
        public DayOfTheWeek previous(){ return THURSDAY; }
        public DayOfTheWeek next(){ return SATURDAY; }
    },
    SATURDAY (6){
        public DayOfTheWeek previous(){ return FRIDAY; }
        public DayOfTheWeek next(){ return SUNDAY; }
    },
    SUNDAY (7){
        public DayOfTheWeek previous(){ return SATURDAY; }
        public DayOfTheWeek next(){ return MONDAY; }
    };

    public abstract DayOfTheWeek previous();
    public abstract DayOfTheWeek next();

    private final int value;

    DayOfTheWeek(int value){
        this.value = value;
    }

    public int getValue(){ return this.value; }
```

```
}

//Example12.java
import java.util.Scanner;
public class Example12{
    public static void main(String args[]){
        Scanner s = new Scanner(System.in);

        System.out.println("Enter a number between 1 & 7: ");
        int val = s.nextInt();

        while(val < 1 || val > 7){
            System.out.println("Try again: ");
            val = s.nextInt();
        }

        DayOfTheWeek days[] = DayOfTheWeek.values();

        for(DayOfTheWeek day: days) {
            if(day.getValue() == val){
                System.out.println("You chose " + day);
                System.out.println("It is preceded by "
+ day.previous());
                System.out.println("It is followed by "
+ day.next());
                break;
            }
        }
    } //main
} //class
```

A sample output from the above program is the same as Example 11.

CHAPTER 20

Data Structures & the java.util Library

This chapter will give an overview of the three basic data structures: a Linked List, a Stack, and a Queue. It then explores the various classes contained in the *java.util* library.

TOPICS

1. <i>Stacks</i>	431
2. <i>Queues</i>	437
3. <i>Linked Lists</i>	442
4. <i>Java.util.Stack</i>	451
5. <i>Java.util.LinkedList</i>	454
6. <i>Java.util.ArrayList</i>	459
7. <i>Java.util.ArrayDeque</i>	463
8. <i>Java.util.Arrays</i>	468
9. <i>Primitive Streams</i>	472

STACKS

The easiest structure to start with is a **stack** of data. The best example that anyone can give for a stack is this: picture that you are at your favorite all-you-can-eat buffet (mmmmmm....). Its time to choose a plate for the salad (well, roast beef), and you notice there are no plates. So, the busboy comes to the rescue. He then places a "stack" of dishes on the holder. Notice what you do next; you take the plate that is at the top of the stack. You don't take the plate at the bottom! You would be a fool to do that.

So, in truth, the first dinner plate on the stack will be the last dinner plate off the stack. Conversely, the last dinner plate on the stack will be the first off the stack. The acronym we use is **LIFO** (Last In, First Out).

As an example below, we have the following stack. The top most item is the only one available, and is the last in. The bottom most item is the first in, and not accessible.

5. Apples ← TOP
4. Oranges
3. Pears
2. Mangos
1. Peaches ← BOTTOM

Stacks & Arrays

In many ways, a stack can be seen as an array. Index zero will be the first element in the stack, and the highest index (or most recent index) will be the last in the stack. We can use a one-dimensional array to represent a stack. Let's see the below example:

0	1	2	3	4	5	6	7	8	9
'a'	'b'	'c'	'd'	'e'	'f'	'g'	'h'	'i'	'j'

Index 0 will be the bottom of the stack while
Index 9 (so far) will be the top of the stack.

The above array will look like:

'j' ← TOP
'i'
'h'

```
'g'  
'f'  
'e'  
'd'  
'c'  
'b'  
'a' ← BOTTOM
```

Let's create our own stack class called *SimpleStack*.

Interface

Every stack has five primary methods, as defined in this interface. Our class will implement this interface.

```
interface Stack {  
    public Object pop();  
    public void push(Object x);  
    public Object peek();  
    public boolean empty();  
    public int size();  
}
```

When we want to add data to the top of the stack, we will **push** data. This will add data to the end of the array. When we want to remove data from the top of the stack, we will **pop** data. This returns the data and removes it entirely from the array. When we want to see what is on top of the stack without removing it, we can **peek** at the data.

The **empty()** method will tell us if the stack is completely empty, meaning it contains no data in its entire length.

The **size()** method will tell us the size of the stack, meaning how many elements of data are contained in it.

Constructors

The *SimpleStack* class begins with its instance variables. We will use the generic *Object* type here to allow for many different types of data. The data array will be of type *Object*. We also have instance variables for the *top* of the stack, which holds the index of

the last item in the stack, as well as *max_size*, which will default to 1000 if we do not declare it via the use of a constructor.

For the one-argument constructor that takes the desired size of the stack, we need to check if the argument is less than or equal to 0. If so, we will throw a *RuntimeException* as the size of the stack must be at least 1.

```
public class SimpleStack implements Stack{
    private Object data[];
    private int top, max_size;

    public SimpleStack() {
        data = new Object[1000];
        top = -1;
        max_size=1000;
    }
    public SimpleStack(int size) {
        if(size <= 0)
            throw new RuntimeException("Size must be at least 1");
        data = new Object[size];
        top = -1;
        max_size = size;
    }
}
```

The *top* variable is set to -1 in both constructors, since there is no data in the stack.

Size() & empty()

The two auxiliary methods are shown here, which return the current size of the stack, as well as telling us if the stack is empty.

```
public int size() { return top + 1; }
public boolean empty() { return top == -1; }
```

Adding (pushing) Data

The *push()* method will simply check if the size of the stack is at its maximum, and if so, throw a new *RuntimeException*, since we can't add data to a full stack. Otherwise, it simply adds the data to the stack at the appropriate index.

```
public void push(Object x) {
    if (size() == max_size)
        throw new RuntimeException("Stack Full");
    data[++top] = x;
}
```

Removing (popping) Data

The *pop()* method will check if the stack is empty, and throw a *RuntimeException* if so, since we cannot remove data from an empty stack. Otherwise, it will return the data at the top of the stack, and decrement the *top* variable, since that piece has been removed.

```
public Object pop() {
    if (empty()) throw new RuntimeException("Stack Empty");
    return data[top--];
}
```

Peeking at Data

The *peek()* method will simply return the data on the top of the stack without changing the position in the stack.

```
public Object peek() {
    if (empty()) throw new RuntimeException("Stack Empty");
    return data[top];
}
```

Printing the Stack

We will overload the *toString()* method so you can simply print out the stack on screen. The loop here will begin at the *top* index and print out each element in the array.

```
public String toString() {
    String ans = "Stack: \n";
    for (int i = top; i >= 0; i--)
        ans += ( data[i] + " -> ");
    return ans;
}
```

```
}
```

The Full Implementation

Here is the complete implementation of the *SimpleStack*:

```
//Stack.java
interface Stack {
    public Object pop();
    public void push(Object x);
    public Object peek();
    public boolean empty();
    public int size();
}

//SimpleStack.java
public class SimpleStack implements Stack{
    private Object data[];
    private int top, max_size;

    public SimpleStack() {
        data = new Object[1000];
        top = -1;
        max_size = 1000;
    }
    public SimpleStack(int size) {
        if(size <= 0)
            throw new RuntimeException("Size must be at least 1");
        data = new Object[size];
        top = -1;
        max_size = size;
    }

    public int size() { return top + 1; }
    public boolean empty() { return top == -1; }

    public void push(Object x) {
        if (size() == max_size)
            throw new RuntimeException("Stack Full");
        data[++top] = x;
    }
    public Object pop() {
        if (empty()) throw new RuntimeException("Stack Empty");
        return data[top--];
    }

    public Object peek() {
```

```

        if (empty()) throw new RuntimeException("Stack Empty");
        return data[top];
    }

    //method for testing purposes
    public String toString() {
        String ans = "Stack: ";
        for (int i = top; i >= 0; i--)
            ans += data[i];
        if(i != 0) ans += " -> ";
        return ans;
    }
} //class

```

EXAMPLE 1: Testing the *SimpleStack*

Here is a program that demonstrates our *SimpleStack*.

```

public class Example1{
    public static void main(String args[]){
        SimpleStack s = new SimpleStack(10);
        char c = 65; //A

        //add items to the stack
        for(int i = 0; i < 10; i++) {
            s.push(c);
            c++;
        }

        System.out.println(s);

        //remove some items
        for(int i = 0; i < 4; i++)
            s.pop();

        //print again
        System.out.println(s);

        //lowercase d
        c = 100;

        //add more items
        for(int i = 0; i < 3; i++) {
            s.push(c);
            c++;
        }
    }
}

```

```

//print again
System.out.println(s);

//pop the stack entirely
int size = s.size();
for(int i = 0; i < size; i++)
    s.pop();

if(s.empty())
    System.out.println("Good bye!");
}//main
} //class

```

The output from running the above program is:

```

Stack: J -> I -> H -> G -> F -> E -> D -> C -> B -> A
Stack: F -> E -> D -> C -> B -> A
Stack: f -> e -> d -> F -> E -> D -> C -> B -> A
Good bye!

```

QUEUES

A great way to think of a queue is a line of people that are waiting to be serviced, whether that's at an amusement park, supermarket, or sporting event. Lines are called **queues**. There are people in front of you, and in back of you. The person in the front is the first served, while the person in the back is the last served. This policy, in Java terms, is called **FIFO** (First In, First Out).

Queues & Arrays

A queue is quite similar to an array, whereas index 0 is the first in line, while index $n-1$ is the last in line. The *SimpleQueue* class will implement the below interface:

```

interface Queue {
    public Object dequeue();
    public void enqueue(Object x);
    public boolean empty();
    public int size();
}

```

The interface here contains two methods we have already seen: `size()` and `empty()`. However, there are two parts to a Queue that make it a Queue: `enqueue()`, which will add data to the queue (or a new person gets in line); and `dequeue()`, which will remove data from the queue (or a person leaves the line).

Constructor

The class begins with the definition of the array, of the generic Object type, in addition to the `front` and `rear` integers, which hold the index of the front of the line and the back of the line. The `size` variable is the current size of the queue, and the `capacity` is the maximum capacity of the queue.

```
public class SimpleQueue implements Queue{
    private Object data[];
    private int front, rear, size, capacity;
```

The first of the two constructors is the default constructor, that will allow a maximum capacity of 1000, declare the array, and set all the remaining variables to 0, since there is no data thus far in the queue.

The second of the constructors will allow you to declare a queue of a certain size. We will again error check, similar to the `SimpleStack`, that the capacity must be at least 1. If it is not, we will throw a new `RuntimeException`.

```
public SimpleQueue() {
    capacity = 1000;
    data = new Object[capacity];
    front = rear = size = 0;
}

public SimpleQueue(int c) {
    if(c <= 0)
        throw new RuntimeException("Capacity must be at least 1");
    capacity = c;
    data = new Object[capacity];
    front = rear = size = 0;
}
```

size() & empty()

As seen before when creating our *SimpleStack*, these methods will check for the queue being empty and for the size of the queue.

```
public int size() { return size; }
public boolean empty() { return size == 0; }
```

Adding Data (Enqueue)

This method will add data to the queue. This method will first check the size, and if it is equal to the capacity, a *RuntimeException* is thrown, since the queue would be full. Otherwise, the data is added to the rear of the queue. If after adding, the *rear* is the capacity, then set the *rear* to 0, since you will not be adding anymore data to the queue. Finally, increment the size.

```
public void enqueue(Object x) {
    if (size() == capacity)
        throw new RuntimeException("Queue Full");
    data[rear++] = x;
    if(rear == capacity) rear = 0;
    size++;
}
```

Removing Data (Dequeue)

This method will remove data from the queue. The return type is not void, since you want to extract the data from the queue. First, this methods checks if the queue is *empty()* and if so, throws a *RuntimeException*, since you cannot remove data from an empty queue. Next, it places the current *front* data in an *answer* variable and checks if the *front* is then at the *capacity*. If so, set the *front* to 0, since you have no more data to remove. Finally, decrement the *size* and return the data.

```
public Object dequeue() {
    if (empty()) throw new RuntimeException("Queue Empty");
    Object answer = data[front++];
    if (front == capacity) front = 0;
    size--;
}
```

```
        return answer;  
    }
```

Printing the Data

Here is an overloaded *toString()* method for printing the queue. The *i* variable will keep the loop running up to the size of the queue. The *j* variable will start at the front of the queue and print the data.

```
public String toString() {  
    int i, j;  
    String ans = "Queue:  ";  
    for (i = 0, j = front; i < size; i++, j++) {  
        if (j == capacity) j = 0;  
        ans += data[j];  
        if(i != size-1)  
            ans += " -> ";  
    }  
    return ans;  
}
```

The Full Implementation

Here is the complete implementation of our *SimpleQueue*.

```
//Queue.java  
interface Queue {  
    public Object dequeue();  
    public void enqueue(Object x);  
    public boolean empty();  
    public int size();  
}  
  
//SimpleQueue.java  
public class SimpleQueue implements Queue{  
    private Object data[];  
    private int front, rear, size, capacity;  
  
    public SimpleQueue() {  
        capacity = 1000;  
        data = new Object[capacity];  
        front = rear = size = 0;  
    }
```

```

public SimpleQueue(int c) {
    if(c <= 0)
        throw new RuntimeException("Capacity must be at least 1");
    capacity = c;
    data = new Object[capacity];
    front = rear = size = 0;
}

public int size() { return size; }
public boolean empty() { return size == 0; }

public void enqueue(Object x) {
    if (size() == capacity)
        throw new RuntimeException("Queue Full");
    data[rear++] = x;
    if(rear == capacity) rear = 0;
    size++;
}
public Object dequeue() {
    if (empty())
        throw new RuntimeException("Queue Empty");
    Object answer = data[front++];
    if (front == capacity) front = 0;
    size--;
    return answer;
}
public String toString() {
    int i, j;
    String ans = "Queue: ";
    for (i = 0, j = front; i < size; i++, j++) {
        if (j == capacity) j = 0;
        ans += data[j];
        if(i != size-1)
            ans += " -> ";
    }
    return ans;
}
} //class

```

EXAMPLE 2: Testing the *SimpleQueue*

Here is a program that demonstrates our *SimpleQueue*.

```

public class Example2{
    public static void main(String args[]){
        SimpleQueue queue = new SimpleQueue(10);
    }
}

```

```

//add some numbers to the queue
for(int i = 0; i < 10; i++) {
    queue.enqueue(i);
}

//print
System.out.println(queue);

//remove the first 5 items
for(int i = 0; i < 5; i++) {
    queue.dequeue();
}

System.out.println(queue);

//add some more items
queue.enqueue(99);
queue.enqueue(888);
queue.enqueue(765);

System.out.println(queue);

//remove items
queue.dequeue();
queue.dequeue();

//print
System.out.println(queue);
} //main
} //class

```

The output from running the above program is:

Array Queue: 0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9

Array Queue: 5 -> 6 -> 7 -> 8 -> 9

Array Queue: 5 -> 6 -> 7 -> 8 -> 9 -> 99 -> 888 -> 765

Array Queue: 7 -> 8 -> 9 -> 99 -> 888 -> 765

LINKED LISTS

In any programming language, you can create something called a linked list. Let's break down the term "linked list." The word list represents having a list of data. For example, you may have a list like this:

Fruits:

1. Apples
2. Oranges
3. Pears
4. Mangos
5. Peaches
- etc ...

Now the term linked. This means that the list will be tied together. In some respects, it may look like this:

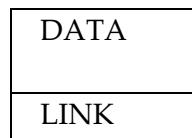
Apples → Oranges → Pears → Mangos → Peaches

Each item in the list is linked to another portion of the list in memory.

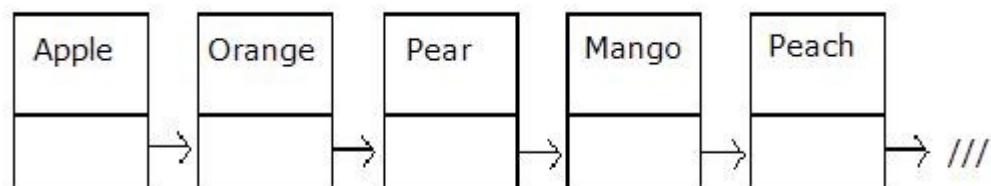
There are two types of linked lists that can be created: **singly linked lists** (which is covered below), and a **doubly linked list**.

Nodes

These are the elements that make it possible to have a linked list. Without them, there would be no listing. Here is how a node looks when you draw a diagram out:



Each node contains two parts: a part for the **data**, and a part for the address, or location, of the next element (or **link**). Here is what the above list will look like when drawing it out:



The last node will always be pointing to nothing (or **null**). As seen, each of the nodes point to each other. Apple points to Orange; Orange points to Pear, etc...

A node for a linked list can be written as a simple class, such as the below:

```
//Node.java
public class Node {
    private Object data;
    private Node next;
    public Node(Object d, Node n) {
        data = d; next = n;
    }
    public Object getData() { return data; }
    public Node getNext() { return next; }
    public void setData(Object d) { data = d; }
    public void setNext(Node n) { next = n; }
}
```

We will be using the `Object` type here, but this can be changed to an appropriate type for your needs. Perhaps you want a linked list of integers, simply change `Object` to `int`.

The constructor will take both the data and the next node as its arguments. The other methods will allow you to access the data (get methods), and set the data (set methods).

Singly Linked Lists

For a singly linked list, all nodes will point in one direction (forward). The last node always points to **null**.

We will begin to build the `LinkedList` class piece by piece and explain it along the way. To begin, the class definition for the `LinkedList` is as follows:

```
public class LinkedList {
    private Node head, tail;
    private int size;
```

Here, we have two nodes: a *head* and a *tail*. This is used for reference points when navigating (or **traversing**) a linked list, as we want to know where it begins and ends. We also have a *size* variable that will simply keep track of how big the list is.

Constructor

```
public LinkedList() {  
    head = tail = null;  
    size = 0;  
}
```

Our constructor simply sets the *head* and *tail* nodes to null, and the *size* to 0, since there is no data added yet.

size() & *empty()*

These methods will check for the list being empty, and for the current size of the list.

```
public int size() { return size; }  
public boolean empty() { return size == 0; }
```

Inserting at the Start (Head)

To insert data to the beginning of the list (the *head* node), we have the following method:

```
public void addHead(Object d) {  
    Node n = new Node(d, head);  
    head = n;  
    size++;  
    if(tail == null) tail = head;  
}
```

We will declare a temporary Node *n*. The new nodes data will be the data from the argument. The node it will point to next is the current *head* node. Finally, assign the *head* node the temporary node *n*, and increase the size. If you do not assign anything to the *head* node, you would not have added anything to the list.

Some more housekeeping applies to the *tail* node. If the *tail* is null, simply make the *tail* point to the *head* node. This would be the case if the list had 0 elements.

Inserting at the End (Tail)

Here, this method will handle adding data to the end of a list.

```
public void addTail(Object d) {  
    Node n = new Node(d, null);  
    if (tail == null) head = tail = n;  
    else {  
        tail.setNext(n);  
        tail = n;  
    }  
    size++;  
}
```

First, declare a temporary Node *n*, where the data it will contain is the data from the argument, and the *next* field points to null (since this is the *tail*, the next field of the *tail* will always point to null).

There are two things that can occur: if the *tail* is null, the list itself is empty, so, therefore, make the *head* and *tail* point to the newly created node. Otherwise, the current *tail*'s next node is the newly created Node *n*, and the *tail* points to that same node. The last thing to do is increase the *size*.

Deleting the Head

The first of the removal method's is to remove the *head* of a list. Here, there is a return type of the method because you may want to print the data out on screen.

```
public Object removeHead() {  
    if (empty()) throw new RuntimeException("Empty List");  
  
    //temporary node is the current head  
    Node n = head;  
    //new head node is the current head's next  
    head = head.getNext();  
  
    if (head == null) tail = head;  
    size--;  
    return n.getData();  
}
```

To start, if the list is empty, throw a *RuntimeException* to let the user know they cannot remove data from an empty list. Next, the temporary Node *n* points to the current *head*. The *head* node will then point to its current *next* node, advancing the *head* by one. If the *head* is now null, make the *tail* point to the *head*, as this means the list is empty. Decrease the *size* and simply return the data from the old *head* node, held in the temporary Node *n*.

Deleting the Tail

The next removal method is to remove the *tail* of a list. Here, there is a return type of the method because you may want to print the data out on screen.

```
public Object removeTail() {
    if (empty()) throw new RuntimeException("Empty List");

    //start at the head
    Node n = head;

    //while you aren't at the tail node, keep going
    while(n.getNext() != tail) {
        n = n.getNext();
    }

    //tail is now the found node
    tail = n;

    //the tail has no next so set to null
    tail.setNext(null);

    //decrease size
    size--;

    return n.getData();
}
```

Again, we need to check if the list is empty. If so, throw a *RuntimeException* to let the user know. Otherwise, we need to navigate to the item in the list that is just before the current *tail* node. Our temporary Node *n* will begin at the current *head* node, and traverse the list until you arrive at the correct node (previous to the *tail*). Once you are there, the current *tail* node is the Node *n*. We then need to set the next node of the *tail* to null, since you are at the end of the list. Decrease the *size* and then return the data.

Traverse the List

When you traverse a linked list, you are simply going through each element of the list. It is easiest to overload the *toString()* method to print out the list:

```
public String toString() {
    String ans = "";
    Node n = head;

    while (n != null) {
        ans += n.getData();
        if (n == tail) ans += " -> ///";
        else ans += " -> ";
        n = n.getNext();
    }
    return ans;
}
```

This simply prints out each element in the list starting at the *head* node.

The Full Implementation

Below is the full implementation of the *LinkedList* we created.

```
//Node.java
public class Node {
    private Object data;
    private Node next;

    public Node(Object d, Node n) {
        data = d; next = n;
    }
    public Object getData() { return data; }
    public Node getNext() { return next; }
    public void setData(Object d) { data = d; }
    public void setNext(Node n) { next = n; }
}

//LinkedList.java
public class LinkedList {
    private Node head, tail;
    private int size;

    public LinkedList() {
```

```

        head = tail = null; size = 0;
    }

    public int size(){ return size; }
    public boolean empty(){ return size == 0; }

    public void addHead(Object d) {
        Node n = new Node(d, head);
        head = n;
        size++;
        if (tail == null) tail = head;
    }

    public void addTail(Object d) {
        Node n = new Node(d, null);
        if (tail == null) head = tail = n;
        else {
            tail.setNext(n);
            tail = n;
        }
        size++;
    }

    public Object removeHead() {
        if (empty())
            throw new RuntimeException("Empty List");

        //temporary node is the current head
        Node n = head;

        //new head node is the current head's next
        head = head.getNext();

        if (head == null) tail = head;
        size--;
        return n.getData();
    }

    public Object removeTail() {
        if (empty())
            throw new RuntimeException("Empty List");

        //start at the head
        Node n = head;

        //while you aren't at the tail node, keep going
        while(n.getNext() != tail) {
            n = n.getNext();
        }

        //tail is now the found node
    }
}

```

```

        tail = n;

        //the tail has no next so set to null
        tail.setNext(null);

        //decrease size
        size--;

        return n.getData();
    }

    public String toString() {
        String ans = "";
        Node n = head;

        while (n != null) {
            ans += n.getData();
            if (n == tail) ans += " -> ///";
            else ans += " -> ";
            n = n.getNext();
        }
        return ans;
    }
} //class

```

EXAMPLE 3: *Testing the LinkedList*

Here is a program that demonstrates our *LinkedList*.

```

public class Example3{
    public static void main(String args[]){
        LinkedList ll = new LinkedList();

        //add items to front
        for(int i = 0; i < 6; i++) {
            ll.addHead(i);
        }

        //add items to end
        for(int i = 7; i < 12; i++) {
            ll.addTail(i);
        }

        System.out.println(ll);

        //remove some items from the front and back
        ll.removeHead();
    }
}

```

```

ll.removeHead();
ll.removeHead();

ll.removeTail();
ll.removeTail();

System.out.println(ll);

//add some items to the end
ll.addTail("Apple");
ll.addTail("Grape");
ll.addTail("Orange");

//print again
System.out.println(ll);

//remove elements from list
int size = ll.size();
for(int i = 0; i < size; i++)
    ll.removeHead();

if(ll.empty()) System.out.println("Good-bye!");
} //main
} //class

```

The output from running the above program is:

```

5 -> 4 -> 3 -> 2 -> 1 -> 0 -> 7 -> 8 -> 9 -> 10 -> 11 -> ///
2 -> 1 -> 0 -> 7 -> 8 -> 9 -> ///
2 -> 1 -> 0 -> 7 -> 8 -> 9 -> Apple -> Grape -> Orange -> ///
Good-bye!

```

JAVA.UTIL.STACK

Surprise! Java has created a *Stack* already that performs the same function as the one we created, but is a little more generic in terms of the data it can use.

In order to use this *Stack*, you must import it from the java.util library.

```
import java.util.Stack;
```

Constructor

The constructor for the *Stack* class creates an empty *Stack*. This declaration does not state what data type is being used.

```
Stack s = new Stack();
```

You may also declare what type of *Stack* you are dealing with. Using the rules of Generics, we will not use any primitive data type.

```
Stack<Integer> s = new Stack<Integer>();
Stack<Double> s = new Stack<Double>();
Stack<String> s = new Stack<String>();
Etc...
```

Member Methods

The methods contained in this *Stack* class are defined below:

boolean empty()

Returns true if the Stack is empty and false if not.

E peek()

This method looks at the data at the top of Stack, without removing it. Throws an EmptyStackException if the Stack is empty.

E pop()

This method returns the data from the top of the Stack and removes it from the Stack. Throws an EmptyStackException if the Stack is empty.

E push(E item)

This method adds data to the top of Stack. It also returns said item.

int search(Object o)

This method returns the position of the argument being searched for in Stack. If it is not on the Stack, the method returns a value of -1.

The below example will showcase java.util.Stack.

EXAMPLE 4: Java Stack Showcase

```
import java.util.Stack;
public class Example4{
    public static void main(String args[]){
        Stack<Integer> s1 = new Stack<Integer>();
        Stack<String> s2 = new Stack<String>();
        char c = 65; //A

        //add items
        for(int i = 0; i < 10; i++) {
            s1.push(i+10);
            s2.push(Character.toString(c++));
        }

        //print the stacks
        System.out.println("S1: " + s1);
        System.out.println("S2: " + s2);

        //remove some data
        s1.pop();
        s1.pop();
        s2.pop();

        //print the stacks
        System.out.println("S1: " + s1);
        System.out.println("S2: " + s2);

        //peek at data
        System.out.println("Top of S1: " + s1.peek());
        System.out.println("Top of S2: " + s2.peek());

        //clear first stack
        while(!s1.empty())
            s1.pop();

        s2.pop();
        s2.pop();

        //print the stacks
        System.out.println("S1: " + s1);
        System.out.println("S2: " + s2);
    }//main
} //class
```

The output from running the above program is:

```
S1: [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
S2: [A, B, C, D, E, F, G, H, I, J]
S1: [10, 11, 12, 13, 14, 15, 16, 17]
S2: [A, B, C, D, E, F, G, H, I]
Top of S1: 17
Top of S2: I
S1: []
S2: [A, B, C, D, E, F, G]
```

JAVA.UTIL.LINKEDLIST

Also contained in the `java.util` library is a *LinkedList*, which functions in its capacity as a doubly linked list.

In order to use this, you must import it from the `java.util` library:

```
import java.util.LinkedList;
```

Constructors

The constructors for the *LinkedList* class will create an empty list, or create one where you can specify the elements with the use of a Collection.

```
public LinkedList()
Default constructor, which creates an empty list.

public LinkedList(Collection<? extends E> c)
The constructor will create a list containing the elements of the
specified Collection.
```

You may also declare what type of *LinkedList* you are dealing with. Using the rules of Generics, we will not use any primitive data type.

```
LinkedList<Integer> ll = new LinkedList<Integer>();
LinkedList<Double> ll = new LinkedList<Double>();
LinkedList<String> ll = new LinkedList<String>();
LinkedList<Float> ll = new LinkedList<Float>();
```

Etc...

Member Methods

The methods contained in this *LinkedList* class are defined below:

boolean add(E element)

Adds the specified parameter to the end of the list. The method also always returns true.

void add(int index, E element)

Adds the specified parameter at the specified index of the list. Throws an *IndexOutOfBoundsException* if the index is out of range.

boolean addAll(Collection <? extends E> c)

Appends all the elements in the specified Collection to the end of this list, in the order they are returned by the specified Collection's iterator. Throws a *NullPointerException* if the specified Collection is null. The method returns true if this list changed as a result of its call.

void addFirst(E element)

Inserts the specified element at the beginning of this list.

void addLast(E element)

Inserts the specified element at the end of this list.

void clear()

This method removes all elements from this list.

Object clone()

This method returns a copy of this *LinkedList*.

boolean contains(Object o)

This method returns true if the specified Object is contained in this list.

E element()

This method returns, but does not remove, the head of this *LinkedList*. Throws a *NoSuchElementException* if this list is empty.

E get(int index)

This method returns the element at the specified index in this *LinkedList*. Throws an *IndexOutOfBoundsException* if the index is out of range.

E getFirst()

This method returns the first element (head) of this `LinkedList`. Throws a `NoSuchElementException` if this list is empty.

E getLast()

This method returns the last element (tail) of this `LinkedList`. Throws a `NoSuchElementException` if this list is empty.

int indexOf(Object o)

This method returns the index of the first occurrence of the specified parameter in the list. It returns -1 if the element is not contained in the list.

int lastIndexOf(Object o)

This method returns the index of the last occurrence of the specified parameter in the list. It returns -1 if the element is not contained in the list.

boolean offer(E element)

This method adds the specified parameter as the last element (tail) of this list. The method always returns true.

boolean offerFirst(E element)

This method inserts the specified parameter at the front (head) of this list. The method always returns true.

boolean offerLast(E element)

This method inserts the specified parameter at the end (tail) of this list. The method always returns true.

E peek()

This method retrieves, but does not remove, the first element of this list or null if the list is empty.

E peekFirst()

This method retrieves, but does not remove, the first element of this list or null if the list is empty.

E peekLast()

This method retrieves, but does not remove, the last element of this list or null if the list is empty.

E poll()

This method retrieves and removes the first element (head) of this list.

E pollFirst()

This method retrieves and removes the first element (head) of this list or null if the list is empty.

E pollLast()

This method retrieves and removes the last element (tail) of this list or null if the list is empty.

E pop()

This method pops an element from the stack represented by this list. Throws a NoSuchElementException if the list is empty.

void push(E e)

This method pushes an element onto the stack represented by this list.

E remove()

This method removes and returns the first element (head) of this list.

E remove(int index)

This method removes and returns the element at the specified index. Throws an IndexOutOfBoundsException if the index is out of range.

boolean remove(Object o)

This method removes the first occurrence of the specified parameter from this list, if it is present. If it is not present, the list remains unchanged. The method returns true if the list contained the specified element.

E removeFirst()

This method removes and returns the first element (head) of this list. Throws a NoSuchElementException if the list is empty.

E removeLast()

This method removes and returns the last element (tail) of this list. Throws a NoSuchElementException if the list is empty.

E removeFirstOccurrence(Object o)

This method removes and returns the first occurrence of the specified parameter in this list, when transversing from head to tail. If the list does not contain the element, the list will be unchanged. The method returns true if the list contained the specified element.

E removeLastOccurrence(Object o)

This method removes and returns the last occurrence of the specified parameter in this list, when transversing from head to tail. If the list does not contain the element, the list will be unchanged. The method returns true if the list contained the specified element.

E set(int index, E element)

This method replaces the element at the specified position in this list with the specified index. The method returns the previous element at the specified index. Throws an IndexOutOfBoundsException if the index is out of range.

```
int size()
```

Returns the number of elements in this list.

The below example will showcase java.util.LinkedList.

EXAMPLE 5: Java *LinkedList* Showcase

```
import java.util.LinkedList;
public class Example5{
    public static void main(String args[]){
        LinkedList<Integer> l1 = new LinkedList<Integer>();
        LinkedList<Integer> l2 = new LinkedList<Integer>();
        LinkedList<Integer> l3;

        //add items
        for(int i = 0; i < 10; i++) {
            l1.addFirst(i+20);
            l2.addLast(i+1);
        }

        //print lists
        System.out.println("L1: " + l1);
        System.out.println("L2: " + l2);

        //work on l3
        l3 = new LinkedList(l2);
        l3.addAll(l1);

        //print l3
        System.out.println(l3);

        //poll data
        System.out.println("L1 head: " + l1.pollFirst());
        System.out.println("L1 tail: " + l1.pollLast());
        System.out.println("L2 head: " + l2.pollFirst());
        System.out.println("L2 tail: " + l2.pollLast());
        System.out.println("L3 head: " + l3.pollFirst());
        System.out.println("L3 tail: " + l3.pollLast());

        System.out.println("L1: " + l1);
        System.out.println("L2: " + l2);
        System.out.println("L3: " + l3);

        l1.offer(100);
        l2.offerFirst(200);
        l3.offerLast(300);
    }
}
```

```

        System.out.println("L1: " + l1);
        System.out.println("L2: " + l2);
        System.out.println("L3: " + l3);

        //check for indices
        System.out.println(l1.indexOf(10));
        System.out.println(l2.indexOf(20));
        System.out.println(l3.indexOf(10));
    } //main
} //class

```

The output from running the above program is:

```

L1: [29, 28, 27, 26, 25, 24, 23, 22, 21, 20]
L2: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20]
L1 head: 29
L1 tail: 20
L2 head: 1
L2 tail: 10
L3 head: 1
L3 tail: 20
L1: [28, 27, 26, 25, 24, 23, 22, 21]
L2: [2, 3, 4, 5, 6, 7, 8, 9]
L3: [2, 3, 4, 5, 6, 7, 8, 9, 10, 29, 28, 27, 26, 25, 24, 23, 22, 21]
L1: [28, 27, 26, 25, 24, 23, 22, 21, 100]
L2: [200, 2, 3, 4, 5, 6, 7, 8, 9]
L3: [2, 3, 4, 5, 6, 7, 8, 9, 10, 29, 28, 27, 26, 25, 24, 23, 22, 21, 300]
-1
-1
8

```

JAVA.UTIL.ARRAYLIST

Also contained in the `java.util` library is an `ArrayList`. An `ArrayList` is essentially a resizable array. This can prove quite useful when dealing with data processing, especially where you may not know the amount of correct data in a text file, and don't want to include any incorrect data in an array. In order to use this, you must import it from the `java.util` library:

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

Constructors

The constructors for the *ArrayList* class creates a default *ArrayList* of size 10; an *ArrayList* of a specified size; and one where you can specify the elements with the use of a Collection.

```
public ArrayList()
```

Default constructor, which creates an empty list of size 10.

```
public ArrayList(int size)
```

One argument constructor, which will create an empty list of the specified size.

```
public ArrayList(Collection<? extends E> c)
```

The constructor will create a list containing the elements of the specified Collection.

You may also declare what type of *ArrayList* you are dealing with. Using the rules of Generics, we will not use any primitive data type.

```
ArrayList<Integer> al = new ArrayList<Integer>();  
ArrayList<Double> al = new ArrayList<Double>();  
ArrayList<String> al = new ArrayList<String>();
```

```
ArrayList<Integer> al = new ArrayList<Integer>(15);  
ArrayList<Double> al = new ArrayList<Double>(100);  
ArrayList<String> al = new ArrayList<String>(6);
```

Etc...

Member Methods

The methods contained in this *ArrayList* class are defined below:

```
boolean add(E element)
```

This method adds the specified element to the end of the list.

void add(int index, E element)

This method inserts the specified element at the specified index in this list. Throws an *IndexOutOfBoundsException* if the specified index is out of range.

boolean addAll(Collection <? extends E> c)

This method adds all of the elements in the specified Collection to the end of the list, in the order that they are returned by the specified Collection's iterator. Throws a *NullPointerException* if the specified Collection is null.

boolean addAll(int index, Collection <? extends E> c)

This method adds all of the elements in the specified Collection to the list, in the order that they are returned by the specified Collection's iterator, beginning at the specified index. It shifts all other elements at the specified index to the right. Throws a *NullPointerException* if the specified Collection is null or an *IndexOutOfBoundsException* if the index is out of range.

void clear()

This method removes all elements from this list.

Object clone()

This method returns a copy of this *ArrayList* instance.

boolean contains(Object o)

This method returns true if the specified Object is contained in this list.

E get(int index)

This method returns the element at the specified index in this list. Throws an *IndexOutOfBoundsException* if the index is out of range.

int indexOf(Object o)

This method returns the index of the first occurrence of the specified parameter in the list. It returns -1 if the element is not contained in the list.

boolean isEmpty()

Returns true if the list is empty and false if not.

int lastIndexOf(Object o)

This method returns the index of the last occurrence of the specified parameter in the list. It returns -1 if the element is not contained in the list.

E remove(int index)

This method removes and returns the element at the specified index. Throws an *IndexOutOfBoundsException* if the index is out of range.

`boolean remove(Object o)`

This method removes the first occurrence of the specified parameter from this list, if it is present. If it is not present, the list remains unchanged. The method returns true if the list contained the specified element.

`E set(int index, E element)`

This method replaces the element at the specified position in this list with the specified index. The method returns the previous element at the specified index. Throws an *IndexOutOfBoundsException* if the index is out of range.

`int size()`

Returns the number of elements in this list.

`void trimToSize()`

This method trims the capacity of this list to be the list's current size.

The below example will showcase the `java.util.ArrayList`.

EXAMPLE 6: Java *ArrayList* Showcase

```
import java.util.ArrayList;
public class Example6{
    public static void main(String args[]){
        ArrayList<Double> l1 = new ArrayList<Double>(10);
        ArrayList<Double> l2 = new ArrayList<Double>();
        ArrayList<Double> l3, l4;

        //add data
        for(double i = 10.0; i < 20.0; i+=1.0) {
            l1.add(i);
            l2.add(10.0-i);
        }

        System.out.println("L1: " + l1);
        System.out.println("L2: " + l2);

        //create new list
        l3 = new ArrayList<Double>();
        l3.addAll(l2);

        System.out.println("L3: " + l3);

        //remove some items
        l1.remove(5);
```

```

12.remove(5);
11.remove(7);
12.remove(7);

//create a new list
14 = new ArrayList<Double>(10);
14.addAll(l1);
14.trimToSize();

//update lists
14.set(2, 500.0);
12.set(2, 400.0);
11.set(5, 900.0);

Double d1, d2;
d1 = 11.get(6);
d2 = 12.get(2);

if(d1 < d2) {
    System.out.println("LESS");
} else {
    System.out.println("MORE");
}

System.out.println("L1: " + l1);
System.out.println("L2: " + l2);
System.out.println("L3: " + l3);
System.out.println("L4: " + l4);
} //main
} //class

```

The output from running the above program is:

L1: [10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0, 19.0]

L2: [0.0, -1.0, -2.0, -3.0, -4.0, -5.0, -6.0, -7.0, -8.0, -9.0]

L3: [0.0, -1.0, -2.0, -3.0, -4.0, -5.0, -6.0, -7.0, -8.0, -9.0]

LESS

L1: [10.0, 11.0, 12.0, 13.0, 14.0, 900.0, 17.0, 19.0]

L2: [0.0, -1.0, 400.0, -3.0, -4.0, -6.0, -7.0, -9.0]

L3: [0.0, -1.0, -2.0, -3.0, -4.0, -5.0, -6.0, -7.0, -8.0, -9.0]

L4: [10.0, 11.0, 500.0, 13.0, 14.0, 16.0, 17.0, 19.0]

JAVA.UTIL.ARRAYDEQUE

Also contained in the `java.util` library is an *ArrayDeque*. An *ArrayDeque* is essentially a Queue (seen above), but allows data to be added and removed to both ends, rather than just the end.

In order to use this, you must import it from the `java.util` library.

```
import java.util.ArrayDeque;
```

Constructors

There are numerous constructors for the *ArrayDeque* class:

public ArrayDeque()

Default constructor, which creates an empty array deque of size 16.

public ArrayDeque(int size)

One argument constructor, which will create an empty array deque of the specified size.

public ArrayDeque(Collection<? extends E> c)

The constructor will create an array deque containing the elements of the specified Collection.

You may also declare what type of *ArrayDeque* you are dealing with. Using the rules of Generics, we will not use any primitive data type.

```
ArrayDeque<Integer> d = new ArrayDeque<Integer>();
```

```
ArrayDeque<String> d = new ArrayDeque<String>();
```

```
ArrayDeque<Integer> d = new ArrayDeque<Integer>(15);
```

```
ArrayDeque<Double> d = new ArrayDeque<Double>(100);
```

Etc...

Member Methods

The methods contained in this *ArrayDeque* class are defined below:

boolean add(E element)

Adds the specified parameter to the end of the deque. Returns true if the element was added successfully in the deque and false otherwise. Throws a NullPointerException if the specified element is null.

void addFirst(E element)

Inserts the specified element at the beginning of this deque. Throws a NullPointerException if the specified element is null.

void addLast(E element)

Inserts the specified element at the end of this deque. Throws a NullPointerException if the specified element is null.

void clear()

This method removes all elements from this deque.

ArrayDeque<E> clone()

This method returns a copy of this deque.

boolean contains(Object o)

This method returns true if the specified Object is contained in this deque.

E element()

This method returns, but does not remove, the first element (head) of this deque. Throws a NoSuchElementException if this deque is empty.

E getFirst()

This method returns the first element (head) of this deque. Throws a NoSuchElementException if this deque is empty.

E getLast()

This method returns the last element (tail) of this deque. Throws a NoSuchElementException if this deque is empty.

boolean isEmpty()

This method returns true if this deque contains no elements and false otherwise.

boolean offer(E element)

This method adds the specified parameter as the last element (tail) of this deque. The method always returns true.

boolean offerFirst(E element)

This method inserts the specified parameter at the front (head) of this deque. The method always returns true.

boolean offerLast(E element)

This method inserts the specified parameter at the end (tail) of this deque. The method always returns true.

E peek()

This method retrieves, but does not remove, the first element of this deque or null if the deque is empty.

E peekFirst()

This method retrieves, but does not remove, the first element of this deque or null if the deque is empty.

E peekLast()

This method retrieves, but does not remove, the last element of this deque or null if the deque is empty.

E poll()

This method retrieves and removes the first element (head) of this deque or null if the deque is empty.

E pollFirst()

This method retrieves and removes the first element (head) of this deque or null if the deque is empty.

E pollLast()

This method retrieves and removes the last element (tail) of this deque or null if the deque is empty.

E pop()

This method pops an element from the stack represented by this deque. Throws a NoSuchElementException if the list is empty.

void push(E e)

This method pushes an element onto the stack represented by this deque.

E remove()

This method removes and returns the first element (head) of this deque.

boolean remove(Object o)

This method removes the first occurrence of the specified parameter from this deque, if it is present. If it is not present, the deque remains unchanged. The method returns true if the deque contained the specified element.

E removeFirst()

This method removes and returns the first element (head) of this deque. Throws a NoSuchElementException if the deque is empty.

E removeLast()

This method removes and returns the last element (tail) of this deque. Throws a NoSuchElementException if the deque is empty.

E removeFirstOccurrence(Object o)

This method removes and returns the first occurrence of the specified parameter in this deque, when transversing from head to tail. If the deque does not contain the element, the deque will be unchanged. The method returns true if the deque contained the specified element.

E removeLastOccurrence(Object o)

This method removes and returns the last occurrence of the specified parameter in this deque, when transversing from head to tail. If the deque does not contain the element, the deque will be unchanged. The method returns true if the deque contained the specified element.

int size()

Returns the number of elements in this deque.

The below example will showcase the java.util.ArrayDeque.

EXAMPLE 7: Java ArrayDeque Showcase

```
import java.util.ArrayDeque;
public class Example7{
    public static void main(String args[]){
        ArrayDeque<Integer> d1 = new ArrayDeque<Integer>(10);
        ArrayDeque<Integer> d2 = new ArrayDeque<Integer>(10);
        ArrayDeque<Integer> d3;

        //add elements
        for(int i = 0; i < 10; i++) {
            if(i % 2 == 0)
                d1.add(i);
            else
                d1.addFirst(i);

            d2.add(i);
        }

        //print deques
        System.out.println(d1);
        System.out.println(d2);

        //remove some data
        d1.remove(0);
        d1.remove(3);
        d2.remove();
        d2.remove(6);

        int size = d1.size();
```

```

//create new deque
d3 = new ArrayDeque<Integer>(size);

//add items to d3
for(int i = 0; i < size; i++) {
    int sum = d1.pollFirst() + d2.pollLast();
    d3.add(sum);
}

//print deque
System.out.println(d1);
System.out.println(d2);
System.out.println(d3);
} //main
} //class

```

The output from running the above program is:

```

[9, 7, 5, 3, 1, 0, 2, 4, 6, 8]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[]
[]
[18, 15, 12, 6, 6, 7, 8, 9]

```

JAVA.UTIL.ARRAYS

Also contained in the java.util library is an *Arrays* library. In order to use this, you must import it from the java.util library:

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

Member Methods

The methods contained in this library are defined below.

Due to the fact that there are many duplicate methods in this library, DT will represent “data type” (int, float, double, long, short, char). In the event of an array for a DT, it will be represented as DT[]. Object data types will be reflected accordingly.

static int binarySearch(DT[] a, DT key)

This method searches the array of the specified parameter for specified key, utilizing a binary search algorithm. The elements of the array must be sorted otherwise the results are undefined. It returns the index of the specified key in the area if it exists and -1 otherwise.

static int binarySearch(Object[] a, Object key)

This method searches the array of the specified parameter for specified key, utilizing a binary search algorithm. The elements of the array must be sorted otherwise the results are undefined. It returns the index of the specified key in the area if it exists and -1 otherwise. Throws a *ClassCastException* if the search key is not comparable to the elements of the array.

static int binarySearch(DT[] a, int fromIndex, int toIndex, DT key)

This method searches the array of the specified parameter for specified key, in a specified range beginning at the **fromIndex** and ending at the **toIndex**, utilizing a binary search algorithm. The elements of the array must be sorted otherwise the results are undefined. It returns the index of the specified key in the area if it exists and -1 otherwise. Throws an *IllegalArgumentException* if **fromIndex** > **toIndex** or an *ArrayIndexOutOfBoundsException* if **fromIndex** < 0 or **toIndex** > **a.length**.

static int binarySearch(Object[] a, int fromIndex, int toIndex, Object key)

This method searches the array of the specified parameter for specified key, in a specified range beginning at the **fromIndex** and ending at the **toIndex**, utilizing a binary search algorithm. The elements of the array must be sorted otherwise the results are undefined. It returns the index of the specified key in the area if it exists and -1 otherwise. Throws a *ClassCastException* if the search key is not comparable to the elements of the array; an *IllegalArgumentException* if **fromIndex** > **toIndex** or an *ArrayIndexOutOfBoundsException* if **fromIndex** < 0 or **toIndex** > **a.length**.

static DT[] copyOf(DT[] original, int newLength)

This method copies the specified parameter, truncating or padding with zeros if needed, so the copy has the specified **newLength**. For all indices that are valid in both the original array and the copy, the two arrays will contain the duplicate values, otherwise the copy will contain zeros. Throws a *NegativeArraySizeException* if **newLength** is negative, or a *NullPointerException* if **original** is null.

static DT[] copyOfRange(DT[] original, int fromIndex, int toIndex)

This method copies the specified parameter, starting at the specified **fromIndex** and ending at the specified **to** index. If the **toIndex** is greater than or equal to the length of the original array, zeros will be padded in the returned array. The length of the returned array will be from **toIndex-fromIndex**. Throws a *ArrayIndexOutOfBoundsException* if

from < 0 or from > original.length; an IllegalStateException if from > to, or a NullPointerException if original is null.

static boolean equals(DT[] a1, DT[] a2)
static boolean equals(Object[] a1, Object[] a2)

Returns true if both specified arrays are equal. An array is considered equal if they contain the same number of elements and all elements are in the same order. Two array references are considered equal if both are null.

static void fill(DT[] a, DT val)

This method assigns the specified val to each element of the specified array.

static void fill(Object[] a, Object val)

This method assigns the specified val to each element of the specified array. Throws an ArrayStoreException if the specified value is not of a runtime type that can be stored in the specified array.

static void fill(DT[] a, int fromIndex, int toIndex, DT val)

*This method assigns the specified val to each element of the specified range in the specified array. The range to be filled extends from **fromIndex** (inclusive) to **toIndex** (exclusive). If fromIndex == toIndex, the range to be filled is empty. Throws an ArrayIndexOutOfBoundsException if fromIndex < 0 or toIndex > a.length, or an IllegalStateException if fromIndex > toIndex.*

static void fill(Object[] a, int fromIndex, int toIndex, Object val)

*This method assigns the specified val to each element of the specified range in the specified array. The range to be filled extends from **fromIndex** (inclusive) to **toIndex** (exclusive). If fromIndex == toIndex, the range to be filled is empty. Throws an ArrayIndexOutOfBoundsException if fromIndex < 0 or toIndex > a.length; an IllegalStateException if fromIndex > toIndex or an ArrayStoreException if the specified value is not of a runtime type that can be stored in the specified array.*

static void parallelSort(DT[] a)

This method sorts the specified array into ascending numerical order using the Parallel Sort algorithm.

static void parallelSort(DT[] a, int fromIndex, int toIndex)

*This method sorts the specified array into ascending numerical order, in the specified range beginning at **fromIndex** (inclusive) and ending at **toIndex** (exclusive), using the Parallel Sort algorithm. Throws an ArrayIndexOutOfBoundsException if fromIndex < 0 or toIndex > a.length, or an IllegalStateException if fromIndex > toIndex.*

static void sort(DT[] a)

This method sorts the specified array into ascending numerical order.

```
static void sort(Object[] a)
```

This method sorts the specified array into ascending order, based on the natural order of its elements.

```
static void sort(DT[] a, int fromIndex, int toIndex)
```

This method sorts the specified array into ascending numerical order, in the specified range beginning at **fromIndex** (inclusive) and ending at **toIndex** (exclusive). Throws an `ArrayIndexOutOfBoundsException` if `fromIndex < 0` or `toIndex > a.length`, or an `IllegalArgumentException` if `fromIndex > toIndex`.

```
static void sort(Object[] a, int fromIndex, int toIndex)
```

This method sorts the specified array into ascending order, based on the natural order of its elements, in the specified range beginning at **fromIndex** (inclusive) and ending at **toIndex** (exclusive). Throws an `ArrayIndexOutOfBoundsException` if `fromIndex < 0` or `toIndex > a.length`; an `IllegalArgumentException` if `fromIndex > toIndex`, or a `ClassCastException` if the array contains elements that are not mutually comparable (for example, `Integer` and `String`).

```
static String toString(DT[] a)
```

This method returns a `String` representation of the specified array.

The below example will showcase `java.util.Arrays`.

EXAMPLE 8: Java Arrays Showcase

```
import java.util.Arrays;
public class Example8{
    private static void print(short arr[]) {
        for(int i = 0; i < arr.length; i++)
            System.out.print(arr[i] + " ");
        System.out.println();
    }
    public static void main(String args[]) {
        short arr1[] = new short[10];
        short arr2[] = {50, 90, 70, 100, 30};

        //fill arr1 with data
        for(short i = 0; i < 10; i++)
            arr1[i] = (short)((10-i)*i);

        print(arr1);
        print(arr2);

        //sort arrays
        Arrays.sort(arr1);
```

```

        Arrays.parallelSort(arr2);

        print(arr1);
        print(arr2);

        //print results of the binary search
        System.out.println(Arrays.binarySearch(arr1, 0, 6, (short)21));
        System.out.println(Arrays.binarySearch(arr2, 1, 3, (short)40));

        //fill arrays
        Arrays.fill(arr1, (short)10);
        Arrays.fill(arr2, 0, 2, (short)5);

        //modify data
        arr2[0] = 101;
        arr1[1] = 11;
        arr1[6] = 2;

        //sort arrays again in certain range
        Arrays.sort(arr1, 0, 3);
        Arrays.sort(arr2, 0, 3);

        print(arr1);
        print(arr2);
    } //main
} //class

```

The output from running the above program is:

```

0 9 16 21 24 25 24 21 16 9
50 90 70 100 30
0 9 9 16 16 21 21 24 24 25
30 50 70 90 100
5
-2
10 10 11 10 10 10 2 10 10 10
5 70 101 90 100

```

PRIMITIVE STREAMS

A new feature introduced in Java 8 is something called a **Primitive Stream**. In this case, they introduced **IntStream**, **DoubleStream**, and **LongStream** for primitive data types int, double, and long.

In order to use any of the above streams, you must import them. It also may be possible and worthwhile to import their Optional counterparts contained in the util library:

```
import java.util.OptionalInt;
import java.util.stream.IntStream;

import java.util.OptionalDouble;
import java.util.stream.DoubleStream;

import java.util.OptionalLong;
import java.util.stream.LongStream;
```

When observing the rest of this section, there are some methods that are a **terminal operation**. This means that no value, whether a primitive value or collection, is returned. The stream can no longer be worked on, nor is it active. There are a few more types of operations and technicalities, but those will not be discussed here.

IntStream & OptionalInt

The IntStream class, and its container OptionalInt, have some useful member methods contained in them.

OptionalDouble average()

This method returns an OptionalDouble describing the arithmetic mean of the elements in this stream, or an empty Optional if this stream is empty. This is a terminal operation.

long count()

Returns the count of the elements in this stream.

IntStream distinct()

This method returns a stream consisting of the distinct elements of the current IntStream.

static IntStream empty()

This method returns an empty IntStream.

OptionalInt findAny()

This method returns an OptionalInt describing some element of this stream, or an empty OptionalInt if the current stream is empty. This is a short-circuiting terminal operation.

OptionalInt findFirst()

This method returns an OptionalInt describing the first element of this stream, or an empty OptionalInt if the current stream is empty. This is a short-circuiting terminal operation.

void forEach(IntConsumer action)

Performs an action for each element of this stream. This is a terminal operation. Also note that there will be a :: in this statement.

IntStream limit(long maxSize)

Returns a stream consisting of the elements of the current IntStream truncated to be no longer than maxSize in length.

OptionalInt min()

OptionalInt max()

These methods returns an OptionalInt describing the maximum or minimum element of this stream, or an empty Optional if this stream is empty. This is a terminal operation.

static IntStream of(int t)

static IntStream of(int ... values)

This method creates a sequential IntStream of a single value t or a series of values, each separated by a comma.

static IntStream range(int startInclusive, int endExclusive)

This method returns a sequential ordered IntStream from startInclusive to endExclusive by an incremental step of 1.

static IntStream rangeClosed(int startInclusive, int endInclusive)

This method returns a sequential ordered IntStream from startInclusive to endInclusive by an incremental step of 1.

IntStream skip(long n)

This method returns an IntStream, disregarding the first n elements of said stream.

IntStream sorted()

This method returns an IntStream consisting of the elements of this stream in sorted order.

int[] toArray()

This method returns the current IntStream in the form of an array. This is a terminal operation.

EXAMPLE 9: *IntStream Showcase*

```
import java.util.stream.IntStream;
public class Example9{
    public static void main(String args[]) {
```

```

IntStream st = IntStream.of(5, 7, 2, 33, 10, 2);
IntStream st2, st3, st4;

//print each value
st.forEach(System.out::println);

//create new stream in range
st2 = IntStream.range(10, 20);
System.out.println("Count: " + st2.count());

//create new stream in range
st3 = IntStream.rangeClosed(10, 25);

//display its average value as a double
System.out.println("Avg: " + st3.average().getAsDouble());

//create new stream in range
st4 = IntStream.rangeClosed(100, 150);
int[] arr = st4.toArray();

//display numbers in range divisible by 5
for(int i = 0; i < arr.length; i++) {
    if(arr[i] % 5 == 0) System.out.print(arr[i] + " ");
}
} //main
} //class

```

The output from running the above program is:

```

5
7
2
33
10
2
Count: 10
Avg: 17.5
100 105 110 115 120 125 130 135 140 145 150

```

LongStream & OptionalLong

The *LongStream* class and its container *OptionalLong* have some useful member methods contained in them.

OptionalDouble average()

This method returns an `OptionalDouble` describing the arithmetic mean of the elements in this stream, or an empty `Optional` if this stream is empty. This is a terminal operation.

long count()

Returns the count of the elements in this stream.

LongStream distinct()

This method returns a stream consisting of the distinct elements of the current `LongStream`. This is a terminal operation.

static LongStream empty()

This method returns an empty `LongStream`.

OptionalLong findAny()

This method returns an `OptionalLong` describing some element of this stream, or an empty `OptionalLong` if the current stream is empty. This is a short-circuiting terminal operation.

OptionalLong findFirst()

This method returns an `OptionalLong` describing the first element of this stream, or an empty `OptionalLong` if the current stream is empty. This is a short-circuiting terminal operation.

void forEach(IntConsumer action)

Performs an action for each element of this stream. This is a terminal operation. Also note that there will be a `::` in this statement.

LongStream limit(long maxSize)

Returns a stream consisting of the elements of the current `LongStream` truncated to be no longer than `maxSize` in length.

OptionalLong min()**OptionalLong max()**

These methods returns an `OptionalLong` describing the maximum or minimum element of this stream, or an empty `Optional` if this stream is empty. This is a terminal operation.

static LongStream of(int t)**static LongStream of(int ... values)**

This method creates a sequential `LongStream` of a single value `t` or a series of values, each separated by a comma.

static LongStream range(long startInclusive, long endExclusive)

This method returns a sequential ordered `LongStream` from `startInclusive` to `endExclusive` by an incremental step of 1.

static LongStream rangeClosed(long startInclusive, long endInclusive)

This method returns a sequential ordered LongStream from startInclusive to endInclusive by an incremental step of 1.

LongStream skip(long n)

This method returns an LongStream, disregarding the first n elements of said stream.

LongStream sorted()

This method returns an LongStream consisting of the elements of this stream in sorted order.

long[] toArray()

This method returns the current LongStream in the form of an array. This is a terminal operation.

EXAMPLE 10: *LongStream Showcase*

```
import java.util.stream.LongStream;
public class Example10{
    public static void main(String args[]) {
        LongStream st = LongStream.of(50, 70, 11, 2233, 1020, 234);
        LongStream st2, st3, st4, st5;

        //print each value
        st.forEach(System.out::println);

        //create new stream in range
        st2 = LongStream.range(1000000, 2000000);

        //display count
        System.out.println("Count: " + st2.count());

        //create new stream in range
        st3 = LongStream.rangeClosed(100, 125);

        //display its average value as a double
        System.out.println("Sum: " + st3.sum());

        //create a new stream in range
        st4 = LongStream.rangeClosed(100, 150);
        long[] arr = st4.toArray();

        //print out numbers divisible by both 3 and 5
        for(int i = 0; i < arr.length; i++) {
            if(arr[i] % 3 == 0 && arr[i] % 5 == 0)
                System.out.print(arr[i] + " ");
        }
    }
}
```

```

    //create new stream and print distinct values
    st5 = LongStream.of(1,1,1,2,6,5,5,6,23,4,5,3,1,1,3);
    st5.distinct().forEach(System.out::println);
} //main
} //class

```

The output from running the above program is:

```

50
70
11
2233
1020
234
Count: 1000000
Sum: 2925
105 120 135 150

```

DoubleStream & OptionalDouble

The DoubleStream class, and its container OptionalDouble, have some useful member methods contained in them.

OptionalDouble average()

This method returns an OptionalDouble describing the arithmetic mean of the elements in this stream, or an empty Optional if this stream is empty. This is a terminal operation.

long count()

Returns the count of the elements in this stream.

DoubleStream distinct()

This method returns a stream consisting of the distinct elements of the current DoubleStream.

static DoubleStream empty()

This method returns an empty DoubleStream.

OptionalDouble findAny()

This method returns an OptionalDouble describing some element of this stream, or an empty OptionalDouble if the current stream is empty. This is a short-circuiting terminal operation.

OptionalDouble findFirst()

This method returns an OptionalDouble describing the first element of this stream, or an empty OptionalDouble if the current stream is empty. This is a short-circuiting terminal operation.

void forEach(DoubleConsumer action)

Performs an action for each element of this stream. This is a terminal operation. This is a terminal operation. Also note that there will be a :: in this statement.

DoubleStream limit(long maxSize)

Returns a stream consisting of the elements of the current DoubleStream truncated to be no longer than maxSize in length.

OptionalDouble min()**OptionalDouble max()**

These methods returns an OptionalDouble describing the maximum or minimum element of this stream, or an empty Optional if this stream is empty. This is a terminal operation.

static DoubleStream of(int t)**static DoubleStream of(int ... values)**

This method creates a sequential DoubleStream of a single value t or a series of values, each separated by a comma.

DoubleStream skip(long n)

This method returns an DoubleStream, disregarding the first n elements of said stream.

DoubleStream sorted()

This method returns an DoubleStream consisting of the elements of this stream in sorted order.

double[] toArray()

This method returns the current DoubleStream in the form of an array. This is a terminal operation.

EXAMPLE 11: DoubleStream Showcase

```
import java.util.OptionalDouble;
import java.util.stream.DoubleStream;
public class Example11{
    public static void main(String args[]) {
        DoubleStream st =
            DoubleStream.of(1.1, 3.3, 2.2, 6.6, 7.7, 5.5, 4.4);

        DoubleStream st2, st3, st4, st5;
```

```

//sort stream and print each value
st.sorted().forEach(System.out::println);

//create new stream and print minimum value as a double
st2 = DoubleStream.of(7, 6, 5, 4, 3, 2, 1.1);

System.out.println("Min: " + st2.min().getAsDouble());

//create new stream and print its count
st3 = DoubleStream.of(5.5555);

System.out.println("Count: " + st3.count());

//create a new stream and print value if its present
st4 = DoubleStream.of(999.999, 777.777, 444.444, 888.888);

OptionalDouble opt = st4.findAny();

if(opt.isPresent()) {
    System.out.println(opt.getAsDouble());
} else {
    System.out.println("654321");
}

//create new stream and print its distinct values
st5 = DoubleStream.of(1, 3, 1, 1, 5, 5, 3, 4, 5, 2, 9);
System.out.println("Distinct: ");

    st5.distinct().forEach(System.out::println);
} //main
} //class

```

The output from running the above program is:

```

1.1
2.2
3.3
4.4
5.5
6.6
7.7
Min: 1.1
Count: 1
999.999
Distinct:

```

1.0
3.0
5.0
4.0
2.0
9.0

CHAPTER 21

Dealing with Dates & Times

This chapter will give an overview of some aspects of the *java.time* library.

TOPICS

1. <i>Month</i>	483
2. <i>Year</i>	486
3. <i>LocalDate</i>	490
4. <i>LocalTime</i>	495
5. <i>LocalDateTime</i>	499
6. <i>DayOfWeek</i>	506
7. <i>Period</i>	509
8. <i>Practical Example: Student Birthdays</i>	513

So far in our Java adventures, we have written programs in prior chapters dealing with dates, and calendars. While that certainly wasn't a waste of time or brain power, we could have utilized some members of the *java.time* library to help us!

There are many libraries contained in the *java.time* package. We will be covering most of them, but not all. This is just a way to show you how programs can handle dates and times as desired for your program(s).

TIME LIBRARY HIGHLIGHTS

Month Class

In order to use this class, you must import it from the *java.time* library:

```
import java.time.Month;
```

This class does not need to be constructed, meaning you do not need to create a new Month object with the use of the new keyword. To create a new instance of the Month object, you can make use of the static **of()** method:

```
Month name = Month.of(int month);
```

Where in the above, **name** is a useful name of the Month object. The parameter of the **of()** method is an int value from 1 to 12, representing the month of the year. This method throws a *DateTimeException* if the given parameter is invalid.

Contained in the Month object are twelve constant, final variables representing the months of the year.

static final Month JANUARY

The singleton instance for the month of January with 31 days. This has a numeric value of 1. Can be accessed from the object as Month.JANUARY;

static final Month FEBRUARY

The singleton instance for the month of February with 28 days (or 29 days in a leap year). This has a numeric value of 2. Can be accessed from the object as Month.FEBRUARY;

static final Month MARCH

The singleton instance for the month of Month with 31 days. This has a numeric value of 3. Can be accessed from the object as Month.MARCH;

static final Month APRIL

The singleton instance for the month of April with 30 days. This has a numeric value of 4. Can be accessed from the object as Month.APRIL;

static final Month MAY

The singleton instance for the month of May with 31 days. This has a numeric value of 5. Can be accessed from the object as Month.MAY;

static final Month JUNE

The singleton instance for the month of June with 30 days. This has a numeric value of 6. Can be accessed from the object as Month.JUNE;

static final Month JULY

The singleton instance for the month of July with 31 days. This has a numeric value of 7. Can be accessed from the object as Month.JULY;

static final Month AUGUST

The singleton instance for the month of August with 31 days. This has a numeric value of 8. Can be accessed from the object as Month.AUGUST;

static final Month SEPTEMBER

The singleton instance for the month of September with 30 days. This has a numeric value of 9. Can be accessed from the object as Month.SEPTEMBER;

static final Month OCTOBER

The singleton instance for the month of October with 31 days. This has a numeric value of 10. Can be accessed from the object as Month.OCTOBER;

static final Month NOVEMBER

The singleton instance for the month of November with 30 days. This has a numeric value of 11. Can be accessed from the object as Month.NOVEMBER;

static final Month DECEMBER

The singleton instance for the month of December with 31 days. This has a numeric value of 12. Can be accessed from the object as Month.DECEMBER;

There are some useful methods contained in the Month class as seen below:

boolean equals(Object other)

Returns true if the specified Month is equal to the current Month and false otherwise.

Month firstMonthOfQuarter()

Returns a Month object, representing the first month of a quarter based on the current Month object. A year can be divided into 4 quarters. If value of the current object is January, February or March, January is returned. If it is April, May or June, April is returned. If it is July, August or September, July is returned. If it is October, November or December, October is returned.

int firstDayOfYear(boolean isLeapYear)

Returns an int value, representing the day of the year corresponding to the first day of the value of the current Month object. The parameter will specify if you are searching for information in a leap year.

int getValue()

Returns an int value, representing the numeric value of the Month object.

int length(boolean isLeapYear)

Returns an int value representing the number of calendar days in the current Month object (from 28 to 31). If parameter is true, it returns 29 for February and 28 otherwise, as February has 29 days in a leap year.

Month minus(long value)

Returns a month of the year that is the specified number of months before the current Month object.

Month plus(long value)

Returns a month of the year that is the specified number of months after the current Month object.

String toString()

Outputs this Month as a String, such as "JANUARY"

Let's see an example showing some of the above methods.

EXAMPLE 1: Month Showcase

```
import java.time.Month;
public class Example1{
    public static void main(String args[]){
        Month month1 = Month.of(5);
        Month month2 = Month.FEBRUARY;
```

```

        System.out.println(month1);
        System.out.println(month2);

        //get int value, representing current months
        System.out.println(month1.getValue());
        System.out.println(month2.getValue());

        //subtract months from current Month objects
        System.out.println(month1.minus(4));
        System.out.println(month2.minus(44));

        //add months from current Month objects
        System.out.println(month1.plus(4));
        System.out.println(month2.plus(44));

        //determine which months are the first of a quarter
        System.out.println(month1.firstMonthOfQuarter());
        System.out.println(month2.firstDayOfYear(false));

        //determine the number of days in a given month
        System.out.println(month1.length(false));
        System.out.println(month2.length(true));
    } //main
} //class

```

The output from running the above program is:

```

MAY
FEBRUARY
5
2
JANUARY
JUNE
SEPTEMBER
OCTOBER
APRIL
32
31
29

```

Year Class

In order to use this class, you must import it from the java.time library:

```
import java.time.Year;
```

This class does not need to be constructed, meaning you do not need to create a new Year object with the use of the new keyword.

To create a new instance of the Year object, you can make use of the **now()** method:

```
Year name = Year.now();
```

Where in the above, **name** is a useful name for the Year object. The now() method will use the system clock to obtain the current year in the default time zone.

To create a new instance of the Year object set to a specific value, you can make use of the static **of()** method:

```
Year name = Year.of(long value);
```

Where in the above, **name** is a useful name of the Year object. The parameter is of type long, which represents the value of the year, such as 2019.

There are some useful methods contained in the Year class as seen below:

boolean equals(Object other)

Returns true if the specified Year object is equal to the current Year object and false otherwise.

int getValue()

Returns an int value, representing the numeric value of the Year object.

boolean isAfter(Year other)

Returns true if the current Year object comes after the specified Year object and false otherwise.

boolean isBefore(Year other)

Returns true if the current Year object comes before the specified Year and false otherwise.

boolean isLeap()

Returns true if the current Year object is a leap year, according to ISO calendar rules, and false otherwise.

boolean isLeap(long year)

Returns true if the parameter is a leap year, according to ISO calendar rules, and false otherwise.

int length()

Returns an int value representing the number of calendar days in the current Year object. This will return 366 if the year is a leap year and 365 otherwise.

Year minusYears(long value)

Returns a copy of the current Year object, minus the number of years specified by the parameter. Throws a DateTimeException if the result exceeds the supported range.

static Year parse(CharacterSequence text)

Returns a Year object parsed from the given text, such as "2019." Throws a DateTimeParseException if the text cannot be parsed.

Year plusYears(long value)

Returns a copy of the current Year object, plus the number of years specified by the parameter. Throws a DateTimeException if the result exceeds the supported range.

String toString()

Outputs this Year as a String, such as "2019"

Let's see an example showing most of the above methods.

EXAMPLE 2: Year Showcase

```
import java.time.Year;
public class Example2{
    public static void main(String args[]){
        Year year1 = Year.now(); //current year
        Year year2 = Year.of(2012); //year 2012

        //print years
        System.out.println(year1);
        System.out.println(year2);

        //check for leap years
        System.out.println(year1.isLeap());
        System.out.println(year2.isLeap());
```

```

//do they match?
System.out.println(year1.equals(year2));

//is year1 after year2?
System.out.println(year1.isAfter(year2));

//is year2 before year1?
System.out.println(year2.isBefore(year1));

//add some years
System.out.println(year1.plusYears(10));
System.out.println(year2.plusYears(2));

//print years again
System.out.println(year1);
System.out.println(year2);

//define new year
Year year3;
year3 = year1.minusYears(4);

//print length of the year
System.out.println(year1.length());
System.out.println(year2.length());
System.out.println(year3.length());

//print final results
System.out.println(year1.getValue());
System.out.println(year2.getValue());
System.out.println(year3.getValue());
} //main
} //class

```

The output from running the above program is:

```

2019
2012
false
true
false
true
true
2029
2014
2019

```

```
2012  
365  
366  
365  
2019  
2012  
2015
```

LocalDate Class

In order to use this class, you must import it from the java.time library:

```
import java.time.LocalDate;
```

This class does not need to be constructed, meaning you do not need to create a new LocalDate object with the use of the new keyword.

This class represents a date and time in the ISO-8601 calendar system, such as 2019-04-02, or more generally (yyyy-MM-dd).

To create a new instance of the LocalDate object, you can make use of the **now()** method:

```
LocalDate name = LocalDate.now();
```

Where in the above, **name** is a useful name for the LocalDate object you need. The now() method will use the system clock to obtain the current date.

If you wanted to create a new instance for a LocalDate, you make use of the various static **of()** methods, such as below, which sets the date to April 2, 2019.

```
LocalDate name = LocalDate.of(2019, 4, 2);
```

The of() method consists of a two overloads:

```
static LocalDate of(int year, int month, int dayOfMonth)
static LocalDate of(int year, Month month, int dayOfMonth)
```

Each of these methods will return a `LocalDate` object representing a specified date and time. The range for the `year` parameter is from `LocalDate.MIN` to `LocalDate.MAX` (below). The range for the numeric `month` is from 1 to 12, representing January to December, respectively. The range for `dayOfMonth` is 1 to 31.

Contained in the `LocalDate` object are two constant, final variables representing the minimum and maximum supported dates. They are of type `LocalDate`:

```
static LocalDate MIN
```

The minimum supported `LocalDate` of "-999999999-01-01"

```
static LocalDate MAX
```

The maximum supported `LocalDate` of "+999999999-12-31"

There are also some useful methods of the `LocalDate` class (`LocalDateTime` will be discussed later in this chapter):

```
LocalDateTime atTime(int hour, int minute)
```

```
LocalDateTime atTime(int hour, int minute, int second)
```

```
LocalDateTime atTime(int hour, int minute, int second, int nanoOfSecond)
```

Combines the current `LocalDate` object with a time to create a `LocalDateTime` object. Range for the `hour` parameter is 0-23. Range for the `minute` parameter is 0-59. Range for `second` parameter is 0-59. Range for `nanoOfSecond` parameter is 0-999,999,999. Throws a `DateTimeException` if any of the parameters are out of range.

```
boolean equals(Object other)
```

Returns true if the specified `LocalDate` is equal to the current `LocalDate` and false otherwise.

```
int getDayOfMonth()
```

Returns an int value (from 1-31) representing the day of the month of the `LocalDate` object.

```
int getDayOfYear()
```

Returns an int value (from 1-366) representing the actual day of the year of the `LocalDate` object.

```
Month getMonth()
```

Returns a Month object representing the Month enum. If value is 1, the method would return JANUARY.

int getMonthValue()

Returns an int value (from 1-12) representing the month of the year of the LocalDate object.

int getYear()

Returns an int value representing the year of the LocalDate object.

boolean isAfter(ChronoLocalDate other)

Returns true if the specified date comes after the current LocalDate object and false otherwise.

boolean isBefore(ChronoLocalDate other)

Returns true if the specified date comes before the current LocalDate object and false otherwise.

boolean isEqual(ChronoLocalDate other)

Returns true if the specified dates are exactly equal and false otherwise.

boolean isLeapYear()

Returns true if the current year value of the LocalDate object is a leap year, according to ISO calendar rules, and false otherwise.

int lengthOfMonth()

Returns an int value (from 1-31) representing the number of days in the month of the LocalDate object.

int lengthOfYear()

Returns an int value representing the number of days in the year of the LocalDate object. It returns 366 if it is a leap year and 365 if not.

LocalDate minusDays(long daysToSubtract)

Returns a copy of the current LocalDate object, minus the number of days specified. Throws a DateTimeException if the result exceeds the supported date range.

LocalDate minusMonths(long monthsToSubtract)

Returns a copy of the current LocalDate object, minus the number of months specified. Throws a DateTimeException if the result exceeds the supported date range.

LocalDate minusYears(long yearsToSubtract)

Returns a copy of the current LocalDate object, minus the number of years specified. Throws a DateTimeException if the result exceeds the supported date range.

LocalDate minusWeeks(long weeksToSubtract)

Returns a copy of the current `LocalDate` object, minus the number of weeks specified. Throws a `DateTimeException` if the result exceeds the supported date range.

static `LocalDate` parse(`CharacterSequence` text)

Returns a `LocalDate` object parsed from the given text, such as "2019-04-02." Throws a `DateTimeParseException` if the text cannot be parsed.

`LocalDate plusDays(long daysToAdd)`

Returns a copy of the current `LocalDate` object, plus the number of days specified. Throws a `DateTimeException` if the result exceeds the supported date range.

`LocalDate plusMonths(long monthsToAdd)`

Returns a copy of the current `LocalDate` object, plus the number of months specified. Throws a `DateTimeException` if the result exceeds the supported date range.

`LocalDate plusYears(long yearsToAdd)`

Returns a copy of the current `LocalDate` object, plus the number of years specified. Throws a `DateTimeException` if the result exceeds the supported date range.

`LocalDate plusWeeks(long weeksToAdd)`

Returns a copy of the current `LocalDate` object, plus the number of weeks specified. Throws a `DateTimeException` if the result exceeds the supported date range.

`String toString()`

Outputs this `LocalDate` as a `String`, such as "2019-04-02"

`LocalDate withDayOfMonth(int dayOfMonth)`

Returns a copy of the current `LocalDate` object, with the day of the month altered. Throws a `DateTimeException` if the `dayOfMonth` value is invalid, or if the `dayOfMonth` is invalid for the month of the year.

`LocalDate withDayOfYear(int dayOfYear)`

Returns a copy of the current `LocalDate` object, with the day of the year altered. Throws a `DateTimeException` if the `dayOfYear` value is invalid, or if the `dayOfYear` is invalid for the year.

`LocalDate withMonth(int month)`

Returns a copy of the current `LocalDate` object, with the month altered. Throws a `DateTimeException` if the month value is invalid.

`LocalDate withYear(int year)`

Returns a copy of the current `LocalDate` object, with the year altered. Throws a `DateTimeException` if the year value is invalid.

Let's see an example showing most of the above methods:

EXAMPLE 3: *LocalDate Showcase*

```
import java.time.LocalDate;
public class Example3{
    public static void main(String args[]){
        LocalDate date = LocalDate.parse("2019-04-02");
        LocalDate date2 = LocalDate.of(2012, 2, 5);

        //print out both dates
        System.out.println(date);
        System.out.println(date2);

        int month, day, year;
        month = date.getMonthValue();
        day = date.getDayOfMonth();
        year = date.getYear();

        //print date1 in another format
        System.out.println(month + "/" + day + "/" + year);

        //print info about date2
        System.out.println(date2.isLeapYear());
        System.out.println(date2.lengthOfMonth());
        System.out.println(date2.lengthOfYear());

        //print info about date with time adjustments
        System.out.println(date.minusWeeks(26));
        System.out.println(date.plusWeeks(26));

        //print info about date2 with time adjustments
        System.out.println(date2.minusDays(40));
        System.out.println(date2.plusDays(40));

        //create a new LocalDate
        LocalDate date3 = date2.withMonth(6);

        //print date3
        System.out.println(date3);
    } //main
} //class
```

The output from running the above program is:

2019-04-02

```
2012-02-05  
4/2/2019  
true  
29  
366  
2018-10-02  
2019-10-01  
2011-12-27  
2012-03-16  
2012-06-05
```

LocalTime Class

In order to use this class, you must import it from the java.time library:

```
import java.time.LocalTime;
```

This class does not need to be constructed, meaning you do not need to create a new LocalTime object with the use of the new keyword. This class represents a date and time in the ISO-8601 calendar system, such as 10:20:30, or more generally (hh:mm:ss.nnn).

To create a new instance of the LocalTime object, you can make use of the **now()** method:

```
LocalTime name = LocalTime.now();
```

Where in the above, **name** is a useful name for the LocalTime object. The now() method will use the system clock to obtain the current time.

If you wanted to create a new instance for a LocalTime, you make use of the various static **of()** methods, such as below, which sets the time to 06:00:30.

```
LocalTime name = LocalTime.of(6, 0, 30);
```

The of() method consists of a few overloads:

```
static LocalTime of(int hour, int minute)
static LocalTime of(int hour, int minute, int second)
static LocalTime of(int hour, int minute, int second, int nanoOfSecond)
Each of these methods will return a LocalTime object representing a
specified time. The range for the hour parameter is 0 to 23. The range
for the minute parameter is 0-59. The range for the second parameter is
0-59. The range for the nanoOfSecond parameter is 0 to 999,999,999.
Throws a DateTimeException if any of the above are out of range.
```

Contained in the LocalDate object are four constant, final variables representing the minimum and maximum supported times, as well as two other constants representing midnight and noon. They are of type LocalTime:

```
static LocalTime MIN
The minimum supported LocalTime of "00:00"

static LocalTime MAX
The maximum supported LocalTime of "23:59:59.999999999"

static LocalTime MIDNIGHT
The time of midnight at the start of a day, "00:00"

static LocalTime NOON
The time of noon in the middle of a day, "12:00"
```

There are some useful methods contained in the LocalTime class as seen below:

```
LocalDateTime atDate(LocalDate date)
Combines the current LocalTime object with a LocalDate to create a
LocalDateTime object.

boolean equals(Object other)
Returns true if the current LocalTime is equal to the specified
LocalTime and false otherwise.

int getHour()
Returns an int value (from 0-23) representing the hour of the LocalTime
object.

int getMinute()
```

Returns an int value (from 0-59) representing the minutes of the LocalTime object.

int getNano()

Returns an int value (from 0-999999999) representing the nanoseconds of the LocalTime object.

int getSecond()

Returns an int value (from 0-59) representing the seconds of the LocalTime object.

boolean isAfter(LocalTime other)

Returns true if the specified time comes after the current LocalTime object and false otherwise.

boolean isBefore(LocalTime other)

Returns true if the specified time comes before the current LocalTime object and false otherwise.

LocalTime minusHours(long hoursToSubtract)

Returns a copy of the current LocalTime object, minus the number of hours specified. Throws a DateTimeException if the result exceeds the supported date range.

LocalTime minusMinutes(long minutesToSubtract)

Returns a copy of the current LocalTime object, minus the number of minutes specified. Throws a DateTimeException if the result exceeds the supported date range.

LocalTime minusNanos(long nanosToSubtract)

Returns a copy of the current LocalTime object, minus the number of nanoseconds specified. Throws a DateTimeException if the result exceeds the supported date range.

LocalTime minusSeconds(long secondsToSubtract)

Returns a copy of the current LocalTime object, minus the number of seconds specified. Throws a DateTimeException if the result exceeds the supported date range.

static LocalTime parse(CharacterSequence text)

Returns a LocalTime object parsed from the given text, such as "10:20:30." Throws a DateTimeParseException if the text cannot be parsed.

LocalTime plusHours(long hoursToAdd)

Returns a copy of the current LocalTime object, plus the number of hours specified. Throws a DateTimeException if the result exceeds the supported date range.

LocalTime plusMinutes(long minutesToAdd)

Returns a copy of the current LocalTime object, plus the number of minutes specified. Throws a DateTimeException if the result exceeds the supported date range.

LocalTime plusNanos(long nanosToAdd)

Returns a copy of the current LocalTime object, plus the number of nanoseconds specified. Throws a DateTimeException if the result exceeds the supported date range.

LocalTime plusSeconds(long secondsToAdd)

Returns a copy of the current LocalTime object, plus the number of seconds specified. Throws a DateTimeException if the result exceeds the supported date range.

String toString()

Outputs this LocalTime as a String, such as "10:20:30"

LocalTime withHour(int hour)

Returns a copy of the current LocalTime object, with the hours altered. Throws a DateTimeException if the hour value is invalid.

LocalTime withMinute(int minute)

Returns a copy of the current LocalTime object, with the minutes altered. Throws a DateTimeException if the minute value is invalid.

LocalTime withNano(int nanoOfSecond)

Returns a copy of the current LocalTime object, with the nanoseconds altered. Throws a DateTimeException if the nanoOfSecond value is invalid.

LocalTime withSecond(int second)

Returns a copy of the current LocalTime object, with the seconds altered. Throws a DateTimeException if the second value is invalid.

EXAMPLE 4: LocalTime Showcase

```
import java.time.LocalTime;
public class Example4{
    public static void main(String args[]){
        LocalTime time1 = LocalTime.of(13, 2, 30);
        LocalTime time2 = LocalTime.of(3, 12, 27);

        System.out.println(time1);
        System.out.println(time2);

        //print some info from each LocalTime
        System.out.println(time1.getHour());
        System.out.println(time1.getMinute())
```

```

        - time2.getMinute());

    //check if times are after each other
    System.out.println(time1.isAfter(time2));
    System.out.println(time2.isAfter(time1));

    //convert the current times over the length of the day
    System.out.println(time1.toSecondOfDay());
    System.out.println(time2.toNanoOfDay());

    //create new LocalTime object
    LocalTime time3 = time1.withHour(3);
    System.out.println(time3.getHour() - time1.getHour());
} //main
} //class

```

The output from running the above program is:

```

13:02:30
03:12:27
13
-10
true
false
46950
11547000000000
-10

```

LocalDateTime Class

In order to use this class, you must import it from the java.time library:

```
import java.time.LocalDateTime;
```

This class does not need to be constructed, meaning you do not need to create a new LocalDateTime object with the use of the new keyword.

This class represents a date and time in the ISO-8601 calendar system, such as 2019-04-02T10:20:30, or more generally (yyyy-MM-ddThh:mm:ss).

To create a new instance of the `LocalDateTime` object, you can make use of the `now()` method:

```
LocalDateTime name = LocalDateTime.now();
```

Where in the above, `name` is a useful name for the `LocalDateTime` object you need. The `now()` method will use the system clock to obtain the current date and time.

If you wanted to create a new instance for a `LocalDateTime`, you make use of the various static `of()` methods, such as below, which will set the date and time to April 2, 2019 at 06:00:30.

```
LocalDateTime name = LocalDateTime.of(2019, 4, 2, 6, 0, 30);
```

The `of()` method consists of a few overloads:

```
static LocalDateTime of(int year, int month, int dayOfMonth, int hour, int minute)
static LocalDateTime of(int year, int month, int dayOfMonth, int hour, int minute, int second)
static LocalDateTime of(int year, int month, int dayOfMonth, int hour, int minute, int second, int nanoOfSecond)
static LocalDateTime of(int year, Month month, int dayOfMonth, int hour, int minute, int second)
static LocalDateTime of(int year, Month month, int dayOfMonth, int hour, int minute, int second, int nanoOfSecond)
```

Each of these methods will return a `LocalDateTime` object representing a specified date and time. The range for the `year` parameter is `LocalDateTime.MIN` to `LocalDateTime.MAX` (see below). The range for the `month` parameter is 1 to 12. The range for the `dayOfMonth` parameter is 1-31. The range for the `hour` parameter is 0 to 23. The range for the `minute` parameter is 0-59. The range for the `second` parameter is 0-59. The range for the `nanoOfSecond` parameter is 0 to 999,999,999. Throws a `DateTimeException` if any of the above are out of range.

Contained in the `LocalDate` object are 2 constant, final variables representing the minimum and maximum supported dates and times. They are of type `LocalDateTime`:

static LocalDateTime MIN

The minimum supported LocalDateTime of “-999999999-01-01T00:00:00”

static LocalDateTime MAX

The maximum supported LocalDateTime of “+999999999-12-31T23:59:59.999999999”

There are some useful methods contained in the LocalDateTime class as seen below:

boolean equals(Object other)

Returns true if the current LocalDateTime object is equal to the specified LocalDateTime and false otherwise.

int getDayOfMonth()

Returns an int value, representing the numeric value of the day of the month in the current object.

DayOfWeek getDayOfWeek()

Returns a DayOfWeek value, representing the enum value of the day of the week (i.e. FRIDAY) in the current object. (DayOfWeek discussed later in the chapter).

int getDayOfYear()

Returns an int value (from 1-366) representing the actual day of the year of the LocalDateTime object.

int getHour()

Returns an int value (from 0-23) representing the hour in the LocalDateTime object.

int getMinute()

Returns an int value (from 0-59) representing the minutes in the LocalDateTime object.

Month getMonth()

Returns a Month object representing the Month enum. If value is 1, the method would return JANUARY.

int getMonthValue()

Returns an int value (from 1-12) representing the month of the year of the LocalDateTime object.

int getNano ()

Returns an int value (from 0-999999999) representing the nanoseconds in the LocalDateTime object.

int getSecond()

Returns an int value (from 0-59) representing the seconds in the LocalDateTime object.

int getYear()

Returns an int value representing the year of the LocalDateTime object.

boolean isAfter(ChronoLocalDateTime other)

Returns true if the current LocalDateTime object comes after the specified LocalDateTime object and false otherwise.

boolean isBefore(ChronoLocalDateTime other)

Returns true if the current LocalDateTime object comes before the specified LocalDateTime object and false otherwise.

boolean isEqual(ChronoLocalDate other)

Returns true if the specified dates and times are exactly equal and false otherwise.

LocalDateTime minusDays(long daysToSubtract)

Returns a copy of the current LocalDateTime object, minus the number of days specified. Throws a DateTimeException if the result exceeds the supported date range.

LocalDateTime minusHours(long hoursToSubtract)

Returns a copy of the current LocalDateTime object, minus the number of hours specified. Throws a DateTimeException if the result exceeds the supported date range.

LocalDateTime minusMinutes(long minutesToSubtract)

Returns a copy of the current LocalDateTime object, minus the number of minutes specified. Throws a DateTimeException if the result exceeds the supported date range.

LocalDateTime minusMonths(long monthsToSubtract)

Returns a copy of the current LocalDateTime object, minus the number of months specified. Throws a DateTimeException if the result exceeds the supported date range.

LocalDateTime minusNanos(long nanosToSubtract)

Returns a copy of the current LocalDateTime object, minus the number of nanoseconds specified. Throws a DateTimeException if the result exceeds the supported date range.

LocalDateTime minusSeconds(long secondsToSubtract)

Returns a copy of the current LocalDateTime object, minus the number of seconds specified. Throws a DateTimeException if the result exceeds the supported date range.

LocalDateTime minusWeeks(long weeksToSubtract)

Returns a copy of the current `LocalDateTime` object, minus the number of weeks specified. Throws a `DateTimeException` if the result exceeds the supported date range.

`LocalDateTime minusYears(long yearsToSubtract)`

Returns a copy of the current `LocalDateTime` object, minus the number of years specified. Throws a `DateTimeException` if the result exceeds the supported date range.

`static LocalDateTime parse(CharacterSequence text)`

Returns a `LocalDateTime` object parsed from the given text, such as "2019-04-02T10:20:30." Throws a `DateTimeParseException` if the text cannot be parsed.

`LocalDateTime plusDays(long daysToAdd)`

Returns a copy of the current `LocalDateTime` object, plus the number of days specified. Throws a `DateTimeException` if the result exceeds the supported date range.

`LocalDateTime plusHours(long hoursToAdd)`

Returns a copy of the current `LocalDateTime` object, plus the number of hours specified. Throws a `DateTimeException` if the result exceeds the supported date range.

`LocalDateTime plusMinutes(long minutesToAdd)`

Returns a copy of the current `LocalDateTime` object, plus the number of minutes specified. Throws a `DateTimeException` if the result exceeds the supported date range.

`LocalDateTime plusMonths(long monthsToAdd)`

Returns a copy of the current `LocalDateTime` object, plus the number of months specified. Throws a `DateTimeException` if the result exceeds the supported date range.

`LocalDateTime plusNanos(long nanosToAdd)`

Returns a copy of the current `LocalDateTime` object, plus the number of nanoseconds specified. Throws a `DateTimeException` if the result exceeds the supported date range.

`LocalDateTime plusSeconds(long secondsToAdd)`

Returns a copy of the current `LocalDateTime` object, plus the number of seconds specified. Throws a `DateTimeException` if the result exceeds the supported date range.

`LocalDateTime plusWeeks(long weeksToAdd)`

Returns a copy of the current `LocalDateTime` object, plus the number of weeks specified. Throws a `DateTimeException` if the result exceeds the supported date range.

`LocalDateTime plusYears(long yearsToAdd)`

Returns a copy of the current `LocalDateTime` object, plus the number of years specified. Throws a `DateTimeException` if the result exceeds the supported date range.

`LocalDate toLocalDate()`

Returns the `LocalDate` portion of this object.

`LocalTime toLocalTime()`

Returns the `LocalTime` portion of this object.

`String toString()`

Outputs this `LocalDateTime` as a `String`, such as "2019-04-02T10:20:30"

`LocalDateTime withDayOfMonth(int dayOfMonth)`

Returns a copy of the current `LocalDateTime` object, with the day of the month altered. Throws a `DateTimeException` if the `dayOfMonth` value is invalid, or if the `dayOfMonth` is invalid for the month-year.

`LocalDateTime withDayOfYear(int dayOfYear)`

Returns a copy of the current `LocalDateTime` object, with the day of the year altered. Throws a `DateTimeException` if the `dayOfYear` value is invalid, or if the `dayOfYear` is invalid for the year.

`LocalDateTime withHour(int hour)`

Returns a copy of the current `LocalDateTime` object, with the hours altered. Throws a `DateTimeException` if the `hour` value is invalid.

`LocalDateTime withMinute(int minute)`

Returns a copy of the current `LocalDateTime` object, with the minutes altered. Throws a `DateTimeException` if the `minute` value is invalid.

`LocalDateTime withMonth(int month)`

Returns a copy of the current `LocalDateTime` object, with the month altered. Throws a `DateTimeException` if the `month` value is invalid.

`LocalDateTime withNano(int nano)`

Returns a copy of the current `LocalDateTime` object, with the nanoseconds altered. Throws a `DateTimeException` if the `nano` value is invalid.

`LocalDateTime withSecond(int second)`

Returns a copy of the current `LocalDateTime` object, with the seconds altered. Throws a `DateTimeException` if the `second` value is invalid.

`LocalDateTime withYear(int year)`

Returns a copy of the current `LocalDateTime` object, with the year altered. Throws a `DateTimeException` if the `year` value is invalid.

Let's see an example of many of the above methods.

EXAMPLE 5: *LocalDateTime Showcase*

```
import java.time.LocalDateTime;
import java.time.Month;
public class Example5{
    public static void main(String args[]){
        //declare LocalDateTime objects
        LocalDateTime dt1 = LocalDateTime.of(
            2019, Month.MARCH, 17, 10, 15, 20);
        LocalDateTime dt2 = LocalDateTime.parse(
            "2015-07-04T11:11:11");

        //print current values
        System.out.println(dt1);
        System.out.println(dt2);

        //extract LocalDate
        System.out.println(dt1.toLocalDate());
        System.out.println(dt2.toLocalDate());

        //extract LocalTime
        System.out.println(dt1.toLocalTime());
        System.out.println(dt2.toLocalTime());

        //check if dates are before each other
        System.out.println(dt1.isBefore(dt2));
        System.out.println(dt2.isBefore(dt1));

        //print out day of the week for each LocalDateTime
        System.out.println(dt1.getDayOfWeek());
        System.out.println(dt2.getDayOfWeek());

        //extract specific values of each LocalDateTime
        int h1, h2, m1, m2, s1, s2, n1, n2;
        h1 = dt1.getHour();
        h2 = dt2.getHour();
        m1 = dt1.getMinute();
        m2 = dt2.getMinute();
        s1 = dt1.getSecond();
        s2 = dt2.getSecond();
        n1 = dt1.getNano();
        n2 = dt2.getNano();

        System.out.println(h1-h2);
        System.out.println(m1+m2);
        System.out.println(s1-s2);
        System.out.println(n1+n2);
    }
}
```

```

//declare new LocalDateTime and print
LocalDateTime dt3 = LocalDateTime.of(
    dt1.minusYears(4).getYear(),
    dt1.minusMonths(1).getMonth(),
    dt1.plusDays(19).getDayOfMonth(), 1, 2, 3);

System.out.println(dt3.toString());
} //main
} //class

```

The output from running the above program is:

```

2019-03-17T10:15:20
2015-07-04T11:11:11
2019-03-17
2015-07-04
10:15:20
11:11:11
false
true
SUNDAY
SATURDAY
-1
26
9
0
2015-02-05T01:02:03

```

DayOfWeek

In order to use this class, you must import it from the java.time library:

```
import java.time.DayOfWeek;
```

This class does not need to be constructed, meaning you do not need to create a new DayOfWeek object with the use of the new keyword.

To create a new instance of the DayOfWeek object, you can make use of the static `of()` method:

```
DayOfWeek name = DayOfWeek.of(int day);
```

Where in the above, `name` is a useful name of the DayOfWeek object. The parameter of the `of()` method is an int value from 1 to 7, representing the day of the week where 1 = Monday and 7 = Sunday. This method throws a `DateTimeException` if the given parameter is invalid.

Contained in the DayOfWeek object are seven constant, final variables representing the days of the week.

static final DayOfWeek MONDAY

The singleton instance for the day of the week known as Monday. This has a numeric value of 1. Can be accessed from the object as `DayOfWeek.MONDAY`;

static final DayOfWeek TUESDAY

The singleton instance for the day of the week known as Tuesday. This has a numeric value of 2. Can be accessed from the object as `DayOfWeek.TUESDAY`;

static final DayOfWeek WEDNESDAY

The singleton instance for the day of the week known as Wednesday. This has a numeric value of 3. Can be accessed from the object as `DayOfWeek.WEDNESDAY`;

static final DayOfWeek THURSDAY

The singleton instance for the day of the week known as Thursday. This has a numeric value of 4. Can be accessed from the object as `DayOfWeek.THURSDAY`;

static final DayOfWeek FRIDAY

The singleton instance for the day of the week known as Friday. This has a numeric value of 5. Can be accessed from the object as `DayOfWeek.FRIDAY`;

static final DayOfWeek SATURDAY

The singleton instance for the day of the week known as Saturday. This has a numeric value of 6. Can be accessed from the object as `DayOfWeek.SATURDAY`;

static final DayOfWeek SUNDAY

The singleton instance for the day of the week known as Sunday. This has a numeric value of 7. Can be accessed from the object as `DayOfWeek.SUNDAY`;

There are some useful methods contained in the `DayOfWeek` class as seen below:

boolean equals(Object other)

Returns true if the current `DayOfWeek` object is equal to the specified `DayOfWeek` object and false otherwise.

static DayOfWeek from(TemporalAccessor temporal)

Returns a `DayOfWeek` object from the specified `temporal`. The parameter should be an object containing a date/time. Throws a `DateTimeException` if unable to convert to a `DayOfWeek`.

int getValue()

Returns an int value, representing the numeric value of the `DayOfWeek` object.

DayOfWeek minus(long value)

Returns a day of the week that is the specified number of days before the current `DayOfWeek` object.

DayOfWeek plus(long value)

Returns a day of the week that is the specified number of days after the current `DayOfWeek` object.

String toString()

Outputs this `DayOfWeek` as a String, such as "SUNDAY"

Let's see an example showing most of the above methods.

EXAMPLE 6: *DayOfWeek Showcase*

```
import java.time.DayOfWeek;
import java.time.LocalDate;
public class Example6{
    public static void main(String args[]){
        DayOfWeek day = DayOfWeek.FRIDAY;
        LocalDate date = LocalDate.parse("2019-02-01");

        System.out.println(day.getValue());
        System.out.println(day.plus(55));
        System.out.println(day.minus(55));
```

```
        day = day.from(date.plusDays(2));
        System.out.println(day.getValue());
        System.out.println(day);
    } //main
} //class
```

The output from running the above program is:

```
5
THURSDAY
SATURDAY
7
SUNDAY
```

Period Class

In order to use this class, you must import it from the java.time library:

```
import java.time.Period;
```

This class does not need to be constructed, meaning you do not need to create a new Period object with the use of the new keyword.

To create a new instance of the Period object, you can make use of the static **of()** method:

```
Period name = Period.of(int years, int months, int days);
```

Where in the above, **name** is a useful name of the Period object. The parameters of the method can be positive or negative, as may want to go forward or backwards in time.

The format of a Period object is most generally (PnYnMnD), representing n number of Years, Months and Day, respectively.

There are also some useful methods contained in the Period class:

```
static Period between(LocalDate dateStartInclusive, LocalDate dateEndInclusive)
```

Returns a Period object representing the number of years, months and days between the two dates specified.

```
boolean equals(Object other)
```

Returns true if the specified Period is equal to the current Period and false otherwise.

```
static Duration from(TemporalAmount temporal)
```

Returns a Duration object from the specified temporal. Throws a DateTimeException if unable to convert to a Period or an ArithmeticException if numeric overflow occurs.

```
int getDays()
```

Returns an int value, representing the numeric value of days contained in this Period object.

```
int getMonths()
```

Returns an int value, representing the numeric value of months contained in this Period object.

```
int getYears()
```

Returns an int value, representing the numeric value of years contained in this Period object.

```
boolean isNegative()
```

Returns true if the current Period is negative in length and false otherwise.

```
boolean isZero()
```

Returns true if the current Period is of zero length and false otherwise.

```
Period minusDays(long days)
```

Returns a copy of this Period object, minus the number of days specified.

```
Period minusMonths(long months)
```

Returns a copy of this Period object, minus the number of months specified.

```
Period minusYears(long years)
```

Returns a copy of this Period object, minus the number of years specified.

```
Period multipliedBy(long multiplicand)
```

Returns a copy of this Period object, multiplied by the scalar specified.

Period negated()

Returns a copy of this Period object with the length negated.

Period normalized()

Returns a copy of this Period object with the years and months normalized. For example, if defined period was 1 year, 15 months, 3 days, the result of this method would be 2 years, 3 months, 3 days.

static Period ofDays(int days)

Creates/obtains a Period object with the specified number of days.

static Period ofMonths(int months)

Creates/obtains a Period object with the specified number of months.

static Period ofYears(int years)

Creates/obtains a Period object with the specified number of years.

static Period parse(CharacterSequence text)

Returns a Period object from the specified text string. Throws a DateTimeParseException if text cannot be parsed.

Period plusDays(long days)

Returns a copy of the Period object with the specified number of days added.

Period plusMonths(long months)

Returns a copy of the Period object with the specified number of months added.

Period plusYears(long years)

Returns a copy of the Period object with the specified number of years added.

String toString()

Outputs this Period as a String, such as "P10Y4M1D"

long toTotalMonths()

Returns an long value representing the total number of months in this Period object.

Period withDays(int days)

Returns a copy of the current Period object, with the number of days altered.

Period withMonths(int months)

Returns a copy of the current Period object, with the number of months altered.

Period withYears(int years)

Returns a copy of the current Period object, with the number of years altered.

Let's see an example showing most of the above methods:

EXAMPLE 7: *Period Showcase*

```
import java.time.Period;
import java.time.LocalDate;
public class Example7{
    public static void main(String args[]){
        //declare LocalDate objects and Period object
        LocalDate date = LocalDate.of(2018, 9, 10); //2019-09-10
        LocalDate date2 = LocalDate.of(1902, 4, 8); //1902-04-08
        Period p;

        //print out LocalDate objects
        System.out.println(date);
        System.out.println(date2);

        //length of time starting at date and ending with date2
        p = Period.between(date, date2);
        System.out.println(p);

        //length of time starting at date2 and ending with date
        p = Period.between(date2, date);
        System.out.println(p);

        //set number of years in the Period and print
        p = Period.ofYears(20000);
        System.out.println(p);

        p = Period.of(5, 18, 5); //5 years, 18 months, 5 days
        System.out.println(p.isNegative());
        System.out.println(p.isZero());

        //normalize the length of the period from above
        p = p.normalized();
        System.out.println(p);
        System.out.println(p.negated());

        //print the LocalDates modified by Period length above
        System.out.println(date.minus(p));
        System.out.println(date2.minus(p));
    } //main
} //class
```

The output from running the above program is:

```
2018-09-10  
1902-04-08  
P-116Y-5M-2D  
P116Y5M2D  
P20000Y  
false  
false  
P6Y6M5D  
P-6Y-6M-5D  
2012-03-05  
1895-10-03
```

PRACTICAL EXAMPLE: STUDENTS' BIRTHDAYS

Pretend you are a teacher and have a class of 10 students. You want to encourage class morale, and celebrate birthdays when they come up. Let's write a program that performs the following tasks:

- 1) Reads from a text file 10 lines of data, each representing a student's information in the following format of name, birthday and gpa:
Alex|1986-09-10|3.5
- 2) The program will load all data into an ArrayList of Student objects.
- 3) Inside the Student object will be instance variables for name, birthday, gpa and birth date of the week.
- 4) Upon loading the data, the program performs some error checking on the birthday and gpa.
 - a. The year specified must be between 1980 and 2000. If it's out of range, a YearException is thrown and message printed to the console.
 - b. The gpa can only be between 0.0 and 4.0. If it's out of range, a GPAException is thrown and message printed to the console.
- 5) If any above errors exist, the program throws the respective Exception, but continues processing the data.
- 6) Once all data is loaded from the file, we want to print (to the console) the following (in order):
 - a. The student with the highest GPA.
 - b. The day of the week each student was born.
 - c. The current age of the student based on the date of 2019-01-01.

Seems like a lot, but let's build this out piece by piece. The text file will look like the following:

```
Alex|1986-09-10|3.5
Ryan|1979-01-02|4.0
Vicky|1999-09-09|3.71
Lisa|1988-08-04|2.2
Jack|2002-11-11|5.2
Frank|1975-03-16|3.34
Robert|1980-01-02|0.0
Mike|1905-10-04|2.3
Dan|1991-07-04|6
Benny|1986-09-10|3.17
```

Judging from the above data, there are 4 invalid dates (data for Ryan, Jack, Frank and Mike) and two invalid GPAs (for Jack and Dan). The program should throw these exceptions when necessary, but continue processing data.

Exceptions

Let's build out the *YearException* and *GPAException* first. They both are going to extend the *IllegalArgumentException* as you are checking if a constructor's argument is valid.

```
//custom Exception
public class YearException extends IllegalArgumentException{
    YearException(String gripe){
        super(gripe);
    }
} //YearException

//custom Exception
public class GPAException extends IllegalArgumentException{
    GPAException(String gripe){
        super(gripe);
    }
} //GPAException
```

Student Class

The Student class will be the main class that contains the information we want to store for the student. It will consist of four private instance variables:

```
private String name;  
private LocalDate birthday;  
private double gpa;  
private DayOfWeek birthday_dayofweek;
```

The constructor will be three arguments: one each for the *name*, *birthday* and *gpa*. The day of the week the student was born can be obtained from the *LocalDate* class.

As stated in the program tasks, we need to do some error checking within the constructor. We want to check if the birthday year is in the range desired (here from 1980-2000). If not, we want to throw the *YearException* we created and continue the program. Similarly with the gpa information, we want to check if the gpa is between 0.0 and 4.0. If it is out of range, we want to throw the *GPAException* and continue the program.

Lastly, we want accessor methods for the data we've stored.

The Student class can be written as the below:

```
import java.time.LocalDate;  
import java.time.DayOfWeek;  
public class Student{  
    //needed variables for student information  
    private String name;  
    private double gpa;  
    private LocalDate birthday;  
    private DayOfWeek birthday_dayofweek;  
  
    //3 argument constructor  
    Student(String n, LocalDate b, double g){  
        //check if GPA is out of range. Valid between 0.0 and 4.0  
        if(g < 0.0 || g > 4.0) {  
            throw new GPAException(  
                "GPA is out of range for student: " + g);  
        }  
        //check if birth year is incorrect.  
        //Valid range is 1980-2000  
        if(b.getYear() < 1980 || b.getYear() > 2000) {  
            throw new YearException(  
                "Birth year is out of range: " + b.getYear());  
        }  
        //otherwise good data so populate variables  
        this.name = n;  
        this.birthday = b;
```

```

        this.gpa = g;
        birthday_dayofweek = birthday.getDayOfWeek();
    }

    //get methods
    public String getName() { return this.name; }
    public LocalDate getBirthday() { return this.birthday; }
    public double getGPA() { return this.gpa; }
    public String getDayOfWeek() { return
                                this.birthday_dayofweek.toString(); }
} //Student

```

Processing Class

The main class called *Processing*, will read the information from the text file and process the data. Since we initially do not know how many rows of data are valid, it is best to use an *ArrayList*, that can be imported from the *java.util* library. It will add items to the list as we go, so long as all data is valid.

Once we have populated the list, we then want to print the student with the highest GPA in the class. That can be done by searching through the list for the index of the highest GPA.

After we have successfully printed the GPA, we then want to print the day of the week the student was born, as well as their age. The age can be determined with the use of the *Period* class, and its member method *between()*, which compares the amount of time that has passed between two dates.

The reason there are two try/catch blocks is that the outer try/catch will be used to catch any Exceptions related to the file reading. The inner try/catch block is contained in the while loop for reading from the file. We want to be able to catch the specific exception we created and continue the program. If any Exceptions are caught, the program executes the finally block of code, which will proceed to the next line of text and continue processing data.

The *Processing* class will look like the below:

```

import java.util.ArrayList;
import java.util.StringTokenizer;
import java.time.LocalDate;
import java.time.Period;

```

```

import java.io.BufferedReader;
import java.io.FileReader;

public class Processing{
    public static void main(String args[]){
        ArrayList <Student> students = new ArrayList<Student> ();

        try {
            BufferedReader br = new BufferedReader(
                new FileReader("students.txt"));

            //read first line
            String line = br.readLine();
            StringTokenizer st;

            //while there is data to process, keep reading
            while(line != null) {
                try {
                    st = new StringTokenizer(line, "|");

                    //first token is the name
                    String name = st.nextToken();

                    //parse the birthday from the file
                    LocalDate birthday =
                        LocalDate.parse(st.nextToken());

                    //get gpa for student
                    double gpa =
                        Double.parseDouble(st.nextToken());

                    //try to build and add Student data
                    Student s = new Student(name, birthday, gpa);

                    //if here, data is good so add to list
                    students.add(s);
                }catch(YearException ye) {
                    System.out.println(ye);
                }catch(GPAException gpaе) {
                    System.out.println(gpaе);
                }finally {
                    //move to next line of data
                    line = br.readLine();
                }
            } //while

            //close file stream
            br.close();

            //used for getting Student info from the list
            Student temp;
        }
    }
}

```

```

        //find max GPA and print
        int index_max = 0;
        for(int i = 1; i < students.size(); i++) {
            temp = students.get(i);
            if(temp.getGPA() >=
                students.get(index_max).getGPA())
                index_max = i;
        }

        //get student who has highest GPA & print
        temp = students.get(index_max);
        System.out.println(temp.getName()
            + " has the highest GPA of " + temp.getGPA());

        //the date to compare the student's birthday to
        LocalDate compare = LocalDate.parse("2019-01-01");
        Period p;

        //print students info by going through the list
        for(int i = 0; i < students.size(); i++) {
            //get student data
            temp = students.get(i);

            //checks the time in between the student's
            //birthday and the date to compare it to.
            p = Period.between(temp.getBirthday(), compare);

            //print information for day of the week for
            //the student and their age. Age is
            //determined by the length of time in
            //between the birthday and compare date

            System.out.println(temp.getName() + " was born on a "
                + temp.getDayOfWeek() + ". They are currently "
                + p.getYears() + " years old.");
        }
    } catch(Exception e) {
        //general exception
        System.out.println(e);
    }

} //main
} //class

```

The output from running the above program is:

YearException: Birth year is out of range: 1979

GPAException: GPA is out of range for student: 5.2

YearException: Birth year is out of range: 1975

YearException: Birth year is out of range: 1905

GPAException: GPA is out of range for student: 6.0

Vicky has the highest GPA of 3.71

Alex was born on a WEDNESDAY. They are currently 32 years old.

Vicky was born on a THURSDAY. They are currently 19 years old.

Lisa was born on a THURSDAY. They are currently 30 years old.

Robert was born on a WEDNESDAY. They are currently 38 years old.

Benny was born on a WEDNESDAY. They are currently 32 years old.

All errors are successfully caught and the program continued populating the ArrayList with valid data.

APPENDICES

Some bonus/reference material can be found in the below Appendices.

1. <i>Appendix A: ASCII Chart</i>	521
2. <i>Appendix B: Number Conversions</i>	522
3. <i>Appendix C: The Game of Keno</i>	535
4. <i>Appendix D: Programming Challenges</i>	565
5. <i>Appendix E: Jeopardy Fun!</i>	575

APPENDIX A

ASCII Chart

ASCII stands for **American Standard Code for Information Interchange**. Please note that the “value” below is the number represented in standard decimal format.

Also note that the characters 1 thru 31 will not be included in the chart as they are considered “control characters,” such as the carriage return, vertical tab, backspace or escape.

Value	Character	Value	Character	Value	Character	Value	Character
32	[SPACE]	57	9	82	R	107	k
33	!	58	:	83	S	108	l
34	"	59	;	84	T	109	m
35	#	60	<	85	U	110	n
36	\$	61	=	86	V	111	o
37	%	62	>	87	W	112	p
38	&	63	?	88	X	113	q
39	'	64	@	89	Y	114	r
40	(65	A	90	Z	115	s
41)	66	B	91	[116	t
42	*	67	C	92	\	117	u
43	+	68	D	93]	118	v
44	,	69	E	94	^	119	w
45	-	70	F	95	_	120	x
46	.	71	G	96	`	121	y
47	/	72	H	97	a	122	z
48	0	73	I	98	b	123	{
49	1	74	J	99	c	124	
50	2	75	K	100	d	125	}
51	3	76	L	101	e	126	~
52	4	77	M	102	f	127	[DEL]
53	5	78	N	103	g		
54	6	79	O	104	h		
55	7	80	P	105	i		
56	8	81	Q	106	j		

There are also “extended” ASCII codes going from 128 to 255 in decimal. These include such characters as the Greek alphabet and mathematical symbols.

APPENDIX B

Number Conversions

There are many number systems in the world of computer science and mathematics. Some systems include the **decimal system** (base ten), **octal system** (base eight), **binary system** (base two), and **hexadecimal system** (base sixteen).

The below chart can certainly come in handy as these are most commonly seen when converting numbers. The table is representing the first 11 powers of 2, which includes 2 to the 0th power.

Power	Result	Binary form
2^0	1	1
2^1	2	10
2^2	4	100
2^3	8	1000
2^4	16	10000
2^5	32	100000
2^6	64	1000000
2^7	128	10000000
2^8	256	100000000
2^9	512	1000000000
2^{10}	1024	10000000000

Representations

Number systems each have different representations. For example, each number we see written out is implied to be in decimal, unless otherwise stated. For example, we could write the following numbers in decimal this way:

1. $(9)_{10}$
2. $(10)_{10}$
3. $(11)_{10}$
- etc...

Each number system has what is called a **base**. As mentioned earlier, decimal is a system of base 10; octal is a system of base 8; binary is a system of base 2; and of course, hexadecimal is a system of base 16. The above numbers in the parentheses show the

base representation for decimal numbers. Either way you write them is acceptable, but it is much more conventional to write them without the parentheses and the base.

The base is also the number of digits or characters used to represent numbers in the system. Below is a chart of the digits or characters each number system uses:

Binary: 0|1

Octal: 0|1|2|3|4|5|6|7

Decimal: 0|1|2|3|4|5|6|7|8|9

Hexadecimal: 0|1|2|3|4|5|6|7|8|9|A|B|C|D|E|F

where A = 10; B = 11; C = 12; D = 13; E = 14; F = 15

So in turn, the range of digits or characters for each system is:

$$0 \leq d < \text{base}$$

Here is a chart showing some decimal numbers and their equivalent representations in each of the mentioned number systems:

Decimal	Binary	Octal	Hexadecimal
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Again, each can be written in the base representation. One example is:

$$(4)_{10} = (100)_2 = (4)_8 = (4)_{16}$$

Conversions

What happens when someone only knows how to read numbers in a certain way? Most everyone knows how to read decimal numbers. So, say we are given a binary number or a hexadecimal number that a person cannot read and understand. We can convert that given number into the appropriate system using a set of rules.

The first type of conversion that is easiest to learn is a binary number to a decimal number.

Binary to Decimal

To take a small example, the binary representation of the decimal number 9 is:

$$(1001)_2$$

The algorithm to go about the conversion is:

- a) Start at the rightmost bit.
- b) Take that bit and multiply by 2^n , where n is the current position beginning at 0 and increasing by 1 each time. This represents a power of two.
- c) Sum each term's product until all bits have been used.

So, using the above methods, here is the resulting conversion to decimal above:

$$\begin{aligned}1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0 \\8 + 0 + 0 + 1 = (9)_{10}\end{aligned}$$

A shortcut is to avoid counting the 0 bits in the sum. It saves space and time when converting, but also remember that you need to increase the exponent for the converting--otherwise the number will not be correct.

Let's look at another example converting the binary number below:

$$(101101001)_2$$

Using the conversion steps above, and skipping the 0 bits, here is the conversion:

$$1 * 2^8 + 1 * 2^6 + 1 * 2^5 + 1 * 2^3 + 1 * 2^0 \\ 256 + 64 + 32 + 8 + 1 = (361)_{10}$$

The best advice for a swift conversion is to practice many times. See the example section at the end of this appendix.

Decimal to Binary

A more annoying and longer way of converting numbers is the decimal system to binary system. The best way of doing this is to follow these steps:

- a) Divide the decimal number by 2.
- b) Take the remainder and record it on the side.
- c) REPEAT UNTIL the decimal number cannot be divided into anymore.
- d) With the bits, record them in order from right to left, as that will be the number in base two.

The example here with the decimal number 8 is a good way to start:

$$\begin{array}{rcl} 8 / 2 & = & 4 \\ 4 / 2 & = & 2 \\ 2 / 2 & = & 1 \\ 1 / 2 & = & 0 \end{array} \quad \begin{array}{r} 0 \\ 0 \\ 0 \\ 1 \end{array}$$

Answer: $(1000)_2$

To check the result, use the previous method of converting from binary to decimal:

$$1 * 2^3 = (8)_{10}$$

This does work, and is an acceptable conversion.

Let's see another example using $(125)_{10}$

$$\begin{array}{rcl} 125 / 2 & = & 62 \\ 62 / 2 & = & 31 \\ 31 / 2 & = & 15 \\ 15 / 2 & = & 7 \end{array} \quad \begin{array}{r} 1 \\ 0 \\ 1 \\ 1 \end{array}$$

$$\begin{array}{r}
 7 / 2 = 3 \\
 3 / 2 = 1 \\
 1 / 2 = 0
 \end{array}
 \quad
 \begin{array}{r}
 1 \\
 1 \\
 1
 \end{array}$$

Answer: $(1111101)_2$

To check the result:

$$\begin{aligned}
 &1 * 2^6 + 1 * 2^5 + 1 * 2^4 + 1 * 2^3 + 1 * 2^2 + 1 * 2^0 \\
 &64 + 32 + 16 + 8 + 4 + 1 = (125)_{10}
 \end{aligned}$$

Binary to Hexadecimal

Another easy conversion to do is binary to hexadecimal. The hexadecimal number system uses the digits 0 to 9 and A, B, C, D, E, F. And, since the hexadecimal system is a power of 2 (2^4), we can take a binary number in groups of 4 and use the appropriate hexadecimal digit in its place.

The steps for doing so are simple. Begin at the rightmost 4 bits. If there are not 4 bits, pad 0s to the left until you hit 4. Repeat the steps until all groups have been converted.

An example is below:

Take the binary number $(1000101)_2$. Using the above steps, here is the conversion:

$$\begin{array}{r}
 0100 \mid 0101 \quad \text{Note that we needed to pad a 0 to the left.} \\
 4 \mid 5
 \end{array}$$

Answer: $(45)_{16}$

Take the binary number $(11101011)_2$. Using the same steps, here is the conversion:

$$\begin{array}{r}
 1110 \mid 1011 \\
 E \mid B
 \end{array}$$

Answer: $(EB)_{16}$

The best advice is to practice each conversion many times with different numbers.

Hexadecimal to Binary

This conversion is also simplistic. Given a hexadecimal number, simply convert each digit to its binary equivalent. Then, combine each 4 bit binary number and that is the resulting answer.

Take the hexadecimal number $(A2F)_{16}$ and convert it to its binary form:

$$\begin{array}{r} A \mid 2 \mid F \\ 1010 \mid 0010 \mid 1111 \end{array}$$

Answer: $(1010\ 0010\ 1111)_2$

Take the hexadecimal number $(55)_{16}$ and convert it to its binary form:

$$\begin{array}{r} 5 \mid 5 \\ 0101 \mid 0101 \end{array}$$

Answer: $(1010101)_2$

Binary to Octal

Since the octal system is again a power of two (2^3), we can take group the bits into groups of 3 and represent each group as an octal digit. The steps are the same for the binary to hexadecimal conversions except we are dealing with the octal base now.

Take the binary number $(10011)_2$ and convert it to octal:

$$\begin{array}{r} 010 \mid 011 \\ 2 \mid 3 \end{array}$$

Answer: $(23)_8$

Take the binary number $(10010101011)_2$ and convert it to octal:

$$\begin{array}{r} 010 \mid 010 \mid 101 \mid 011 \\ 2 \mid 2 \mid 5 \mid 3 \end{array}$$

Answer: $(2253)_8$

Octal to Binary

To go from octal to binary, simply reverse the above algorithm and represent each octal digit in its three bit binary form.

Take the octal number $(742)_8$ and convert it to binary:

$$\begin{array}{r} 7 \mid 4 \mid 2 \\ 111 \mid 100 \mid 010 \end{array}$$

Answer: $(111100010)_2$

Take the octal number $(1234567)_8$ and convert it to binary:

$$\begin{array}{r} 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \\ 001 \mid 010 \mid 011 \mid 100 \mid 101 \mid 110 \mid 111 \end{array}$$

Answer: $(001010011100101110111)_2$

Decimal to Octal

Similar to the decimal to binary method, you must divide each number by 8 and record the remainder. This continues until the number cannot be divided into anymore.

Take the decimal number $(18)_{10}$ and convert it to octal:

$$\begin{array}{r} 18 / 8 = 2 & 2 \\ 2 / 8 = 0 & 2 \end{array}$$

Answer $(22)_8$

Take the decimal number $(46)_{10}$ and convert it to octal:

$$\begin{array}{r} 46 / 8 = 5 & 6 \\ 5 / 8 = 0 & 5 \end{array}$$

Answer: $(56)_8$

Octal to Decimal

Similar to the binary to decimal method, simply take each digit in the octal base and multiply by the power of 8.

Take the octal number $(764)_8$ and convert it to decimal:

$$\begin{aligned}7 * 8^2 + 6 * 8^1 + 4 * 8^0 \\448 + 48 + 4 = 500\end{aligned}$$

Answer: $(500)_{10}$

Take the octal number $(5771)_8$ and convert it to decimal:

$$\begin{aligned}5 * 8^3 + 7 * 8^2 + 7 * 8^1 + 1 * 8^0 \\2560 + 448 + 56 + 1 = 3,065\end{aligned}$$

Answer: $(3065)_{10}$

Again, the best advice is to just practice. It takes time to get used to conversions.

Integer Wrapper Class

With the help of the Integer wrapper class, we can call upon a member method to convert a given number in a particular base; after which, we can convert that result to another base. We can also convert a given base 10 number (decimal) to a binary, octal, and hexadecimal String.

Integer.parseInt(String val, int base)

Returns an int value after converting the String representation of a number in a particular base.

Integer.toString(String val, int base)

Returns a String representation of a given number in a given base.

Integer.toBinaryString(int i)

Returns a String representation of the conversion of a base 10 decimal number to its binary form (base 2).

Integer.toOctalString(int i)

Returns a String representation of the conversion of a base 10 decimal number to its octal form (base 8).

Integer.toHexString(int i)

Returns a String representation of the conversion of a base 10 decimal number to its hexadecimal form (base 16).

Using a combination of the above methods, we can convert any number in any base to another base.

EXAMPLE 1: Simple Number Conversions

The program will ask the user to input a number in base 10 and will print out its binary, octal, and hexadecimal equivalents.

```
import java.util.Scanner;
public class Example1{
    public static void main(String args[]) {
        Scanner s = new Scanner(System.in);
        int entry = 0;

        //prompt user for entry
        System.out.println("Enter a number (in base 10): ");
        entry = s.nextInt();

        //print out Strings
        System.out.println("Binary: "
            + Integer.toBinaryString(entry));
        System.out.println("Octal: "
            + Integer.toOctalString(entry));
        System.out.println("Hexadecimal: "
            + Integer.toHexString(entry));

    } //main
} //class
```

A sample output of the above program is:

Enter a number (in base 10):

15

Binary: 1111

Octal: 17

Hexadecimal: f

Another sample run is:

Enter a number (in base 10):

123456

Binary: 11110001001000000

Octal: 361100

Hexadecimal: 1e240

EXAMPLE 2: More Complex Number Conversions

The program will demonstrate some number conversions using a series of System.out statements.

```
public class Example2{
    public static void main(String args[]) {
        String nums[] = new String[3];
        String convert = "";
        int t = 0;

        nums[0] = "125"; //decimal
        nums[1] = "1AF"; //hexadecimal
        nums[2] = "33"; //octal

        //dealing with nums[0], currently in decimal
        t = Integer.parseInt(nums[0], 10);
        convert = Integer.toString(t, 2); //to binary
        System.out.println("Decimal " + nums[0]
                           + " to binary: " + convert);

        convert = Integer.toString(t, 8); //to octal
        System.out.println("Decimal " + nums[0]
                           + " to octal: " + convert);

        convert = Integer.toString(t, 16); //to hexadecimal
        System.out.println("Decimal " + nums[0]
                           + " to hexadecimal: " + convert);

        //dealing with nums[1], currently in hexadecimal
        t = Integer.parseInt(nums[1], 16);
        convert = Integer.toString(t, 2); //to binary
        System.out.println("Hexadecimal " + nums[1]
                           + " to binary: " + convert);
```

```

convert = Integer.toString(t, 8); //to octal
System.out.println("Hexadecimal " + nums[1]
+ " to octal: " + convert);

convert = Integer.toString(t, 10); //to hexadecimal
System.out.println("Hexadecimal " + nums[1]
+ " to decimal: " + convert);

//dealing with nums[2], currently in octal
t = Integer.parseInt(nums[2], 8);
convert = Integer.toString(t, 2); //to binary
System.out.println("Octal " + nums[2]
+ " to binary: " + convert);

convert = Integer.toString(t, 10); //to decimal
System.out.println("Octal " + nums[2]
+ " to decimal: " + convert);

convert = Integer.toString(t, 16); //to hexadecimal
System.out.println("Octal " + nums[2]
+ " to hexadecimal: " + convert);
} //main
} //class

```

The output from running the above program is:

```

Decimal 125 to binary: 1111101
Decimal 125 to octal: 175
Decimal 125 to hexadecimal: 7d
Hexadecimal 1AF to binary: 110101111
Hexadecimal 1AF to octal: 657
Hexadecimal 1AF to decimal: 431
Octal 33 to binary: 11011
Octal 33 to decimal: 27
Octal 33 to hexadecimal: 1b

```

EXERCISES

DIRECTIONS: For each question, convert the number to the desired base.

1. Convert each given number to binary form.

- a) $(19)_{10}$
- b) $(34)_{10}$

- c) $(100)_{10}$
- d) $(9214)_{16}$
- e) $(9999)_{16}$
- f) $(123)_8$

2. Convert each given number to octal form.

- a) $(1010111)_2$
- b) $(44)_{16}$
- c) $(11110)_2$
- d) $(10001)_2$
- e) $(F24A)_{16}$
- f) $(10101011111001)_2$

3. Convert each given number to hexadecimal form.

- a) $(101011110010101001)_2$
- b) $(1010)_8$
- c) $(734)_8$
- d) $(111011)_2$
- e) $(56)_{10}$
- f) $(170)_{10}$

Solutions to Exercises

1.

- a) $(10011)_2$
- b) $(100010)_2$
- c) $(1100100)_2$
- d) $(1001001000010100)_2$
- e) $(1001100110011001)_2$
- f) $(1010011)_2$

2.

- a) $(127)_8$
- b) $(104)_8$
- c) $(36)_8$
- d) $(21)_8$
- e) $(171112)_8$
- f) $(52771)_8$

3.

- a) $(15F2A9)_{16}$
- b) $(208)_{16}$
- c) $(1DA)_{16}$
- d) $(3B)_{16}$
- e) $(38)_{16}$
- f) $(AA)_{16}$

APPENDIX C

The Game of Keno

For those who have been to a casino or played a state lottery that has this game, we are going to create a console based program that will simulate a game of Keno.

The rules of the game are fairly straightforward. There are numbers 1 to 80 that are in the field. In any one game, 20 numbers are randomly drawn from the field. A player has the option to play anywhere from 1 to 10 numbers (or **spots** as they are called). The more numbers you match, the more you will win, based on the number of spots you play. That's the basic aspect of the game. But how much do you win per game?

Below is a pay table that we will be using for this game which is based on the New York state lottery. Each state's or casino's pay table will vary slightly.

MATCHES	PRIZE	MATCHES	PRIZE
10 SPOT GAME		6 SPOT GAME	
10	\$100,000	6	\$1,000
9	\$5,000	5	\$55
8	\$300	4	\$6
7	\$45	3	\$1
6	\$10	5 SPOT GAME	
5	\$2	5	\$300
0	\$5	4	\$20
9 SPOT GAME		3	\$2
9	\$30,000	4 SPOT GAME	
8	\$3,000	4	\$55
7	\$125	3	\$5
6	\$20	2	\$1
5	\$5	3 SPOT GAME	
0	\$2	3	\$23
8 SPOT GAME		2	\$2
8	\$10,000	2 SPOT GAME	
7	\$550	2	\$10
6	\$75	1 SPOT GAME	
5	\$6	1	\$2
0	\$2	*All payouts above are based on a \$1 wager.	
7 SPOT GAME			
7	\$5,000		
6	\$100		
5	\$20		
4	\$2		
0	\$1		

If you have arrived here from reading the book start to finish, this program incorporates a touch of most topics covered. To name a few, there will be inheritance, Random numbers, the use of arrays, custom exceptions, loops, Scanners, LocalTime, etc.

BUILDING THE PROGRAM

A few ground rules and directions for the game.

1. We want to create two custom Exceptions, named *BankrollException* and *NameException*. They will both extend *IllegalArgumentException*.
2. When playing the game, we will loop continuously until the user enters E to end the program or any Exception is thrown.

We want to break the game up into multiple java files. The below will breakdown what content is included in each, as well as some direction.

NameException.java

Custom exception class, reflecting an error with a Player name.

Imports

[None]

Extends

IllegalArgumentException

Constructor

public NameException(String gripe);

-Will invoke super class constructor, passing the *gripe* argument.

Implementation

```
public class NameException extends IllegalArgumentException{
    public NameException(String gripe) {
        super(gripe);
    } //constructor
} //NameException
```

BankrollException.java

Custom exception class, reflecting an error with a Player bankroll.

Imports

[None]

Extends

IllegalArgumentException

Constructor

public BankrollException(String gripe);

-Will invoke super class constructor, passing the *gripe* argument.

Implementation

```
public class BankrollException extends IllegalArgumentException{
    public BankrollException(String gripe) {
        super(gripe);
    } //constructor
} //BankrollException
```

PastNums.java

Class used to hold information on the previously drawn 100 games.

Imports

import java.util.Random;

Instance variables

public int drawn_times[], drawn_indexes[];

Methods (header lines)

public void sim100();

-Will update *drawn_times* array with information on the 100 past drawn games.

Constructor

public PastNums();

-Will initialize each array and invoke the *sim100()* method.

Implementation

```

import java.util.Random;

/**
 * Super class of Grid.java. Will contain information
 * on the 100 past games before user plays. This is
 * used to hold information for hot and cold number
 * determinations.
 *
 * @author Alex Maureau
 *
 */
public class PastNums{

    //public past drawn numbers and parallel array to hold indexes
    public int drawn_times[], drawn_index[];

    /**
     * Constructor. Will create new arrays and fill information
     * via the use of the sim100() method
     */
    public PastNums(){
        drawn_times = new int[81];
        drawn_index = new int[81];

        //fill arrays
        for(int i = 1; i < 81; i++) {
            drawn_times[i] = 0;
            drawn_index[i] = i;
        }

        //simulate last 100 draws
        sim100();
    }
    /**
     * Method to simulate last 100 games to help determine
     * hot and cold numbers
     */
    public void sim100() {
        Random r = new Random();
        //run 100 past games of numbers
        for(int i = 0; i < 100; i++) {
            //20 numbers are drawn in a game
            for(int j = 1; j <= 20; j++) {
                drawn_times[ r.nextInt(80) + 1]++;
            } //j
        } //i
    } //sim100
} //class

```

Grid.java

Class to hold information on the grid of numbers (1 to 80).

Imports

```
import java.util.Random;
```

Extends

```
PastNums
```

Instance variables

```
public int grid[];
```

Methods (header lines)

```
public void shuffle();
```

-Randomly shuffles the *grid* of numbers. The first 20 numbers in the *grid* after the shuffle are the ones being drawn for the next game.

```
public void printGrid();
```

-Will print a *grid* of numbers to the console, consisting of 8 rows and 10 columns. If a number was drawn, print a “—” to indicate a drawn number, else print the number.

```
private boolean wasDrawn(int num);
```

-Method to determine if a number was drawn in a particular game. If so, return true, else false. Method used with *printGrid()*.

Constructor

```
public Grid();
```

-Will invoke super class constructor first. It then creates a new *grid* of size 81, to allow for use of the indexes 1-80, a reflection of the grid of numbers.

Implementation

```
import java.util.Random;

/**
 * Class to hold information for the grid of 80 numbers.
 * Extends PastNums, to allow for previously drawn numbers
 *
 * @author Alex Maureau
 */

```

```

class Grid extends PastNums{
    //grid of numbers
    public int grid[];

    /**
     * Constructor. Invokes PastNums constructor first
     * and creates a new grid of numbers.
     */
    public Grid(){
        super();

        //to allow for actual numbers 1-80
        grid = new int[81];

        //loop through to set all to loop counter
        for(int i = 1; i < 81; i++) {
            grid[i] = i;
        }
    }

    /**
     * Randomly shuffles the grid of numbers.
     * The first 20 numbers in the grid after the shuffle
     * are the ones being drawn for the next game.
     */
    public void shuffle() {
        int n = grid.length, temp = 0;
        Random random = new Random();
        random.nextInt();
        for (int i = 1; i < n; i++) {
            int change = i + random.nextInt(n - i);
            temp = grid[i];
            grid[i] = grid[change];
            grid[change] = temp;
        }

        //first 20 numbers in the array are drawn so update totals
        for(int i = 1; i <= 20; i++)
            super.drawn_times[i]++;
    }

    /**
     * Method to determine if a number was drawn in a particular game
     * when printing the grid to the console.
     *
     * If so, return true, else false.
     *
     * @param num the number to check
     * @return true if num was drawn in a game
     */
    private boolean wasDrawn(int num) {

```

```

        boolean drawn = false;
        for(int i = 1; i <= 20; i++)
            if(grid[i] == num) {
                drawn = true;
                break;
            }
        return drawn;
    }

    /**
     * Method to determine if a player's number was drawn in
     * a particular game when drawing the numbers.
     *
     * If so, return true, else false.
     *
     * @param num the number to check
     * @param entries the array of numbers the player has in a game
     * @return true if num matches a number in entries
     */
    private boolean wasDrawnEntry(int num, int[] entries) {
        boolean drawn = false;
        for(int i = 0; i < entries.length; i++)
            if(entries[i] == num) {
                drawn = true;
                break;
            }
        return drawn;
    }

    /**
     * Will print a grid of numbers to the console, consisting of 8
     * rows and 10 columns. If a number was drawn, print a "--" to
     * indicate a drawn number, else print the number.
     */
    public void printGrid(int[] entries) {
        System.out.println("Numbers drawn: ");

        try {
            //draw the 20 numbers. use sleep method of Thread
            //to separate the draw time.
            for(int i = 1; i <= 20; i++) {
                Thread.sleep(1000);

                //check if current drawn number is one of
                //players numbers. if so, print an * to
                //denote a matched numbers
                //otherwise just print the number

                if(wasDrawnEntry(grid[i], entries))
                    System.out.print(grid[i] + "* ");
                else

```

```

        System.out.print(grid[i] + " ");
    }
} catch(Exception e) {
    //possible that Thread is interrupted.
    System.out.println(e);
} finally {
    System.out.println();

    //print the grid. the position number is determined
    //by 10*r + c-10

    int pos_num = 0;
    for(int r = 1; r <= 8; r++) {
        for(int c = 1; c <= 10; c++) {
            pos_num = (10*r) + (c-10);

            //if number was drawn, print the "--"
            if(wasDrawn(pos_num)) {
                System.out.print("--\t");
            } else {
                //otherwise print the number
                System.out.print(pos_num + "\t");
            }
        }
        System.out.println();
    } //for
} //finally
} //grid
} //class

```

Player.java

Class to hold information on the Player (name and bankroll).

Imports

[None]

Instance variables

```
private int bankroll, starting_bankroll;
private String name;
```

Methods (header lines)

```
public String toString();
```

-Overloaded `toString()` method, printing the player *name* and current *bankroll* amount.

```
public int getBankroll();
```

-Returns an int representing the current bankroll for the player.

```
public int getStartingBankroll();
```

-Returns an int representing the starting bankroll for the player.

```
public void setBankroll(int b);
```

-Adjusts the current bankroll based on the argument.

Constructor

```
public Player(String n, int b);
```

-Two arguments, representing name (*n*) and starting bankroll (*b*). Performs error checking on the arguments and throws a respective Exception should any issue occur.

Implementation

```
/**  
 * Class to hold name and bankroll information for the user  
 *  
 * @author Alex Maureau  
 *  
 */  
public class Player{  
    //instance variables  
    private int bankroll, starting_bankroll;  
    private String name;  
  
    /**  
     * Two argument constructor for name and bankroll  
     * @param n represents the name of the user  
     * @param b represents the starting bankroll of the user  
     */  
    public Player(String n, int b){  
        //if bankroll less than 0, error  
        if(b <= 0) {  
            throw new BankrollException("No money entered!");  
        }  
        //if name is null or 0 length, error  
        if(n.length() == 0 || n == null) {  
            throw new NameException("Name entry error!");  
        }  
        //good data  
        name = n;  
        bankroll = b;  
        starting_bankroll = bankroll;
```

```

    }

    //get/set method for bankroll.
    public int getBankroll() { return bankroll; }
    public int getStartingBankroll() { return starting_bankroll; }
    public void setBankroll(int amt) {
        bankroll += amt;
    }

    //overloaded toString method, printing the name and
    //remaining bankroll of the user
    public String toString() {
        return name + ", you have $" + bankroll + " left";
    }
} //Player

```

Game.java

Class that contains information for playing a game.

Imports

```

import java.util.Random;
import java.util.Scanner;
import java.time.LocalTime;

```

Instance variables

```

private Grid grid;
private int entries[];
private int nums_played, games_played, pergame_played, total_wager,
game_id;
private LocalTime time;

```

Methods (header lines)

```
public int getGamesPlayed();
```

-Returns a value representing the number of consecutive games being played.

```
public int getTotalWager();
```

-Returns a value representing the total amount of the wager.

```
private boolean isDrawn(int n);
```

-Method to determine if the argument *n* was drawn in the *entries* array. Used primarily when generating quick pick numbers.

```
private void printEntries();
```

-Method to print the entries the player has for a game.

```
private void firstGame(Player p);
```

-Method to prompt the player for the first time when creating a game. It will ask for the numbers desired to play; whether they want a quick pick of numbers; how many consecutive games they want to play; and how much they are wagering per game. Will update the bankroll of the Player p assuming no exception is thrown.

```
public void play(boolean played_game, Player p);
```

-Method to play a game. If $played_game$ is true, then we do not need to setup a new game. This method will generate the $game_id$, print the players' *entries*, shuffle the numbers in the $grid$, draw the numbers for the game and finally print the $grid$.

```
public void hotNumbers();
```

-Method to determine and print the 10 hottest numbers (most frequently drawn).

```
public void coldNumbers();
```

-Method to determine and print the 10 coldest numbers (least frequently drawn).

```
public int checkWin();
```

-Method to determine the total amount of the win after a played game. It will first determine the number of matches the player has in a game, and will then invoke the $payTable()$ method, where we get the actual amount of the win based on the number of matches. This method does return the total win.

```
private int payTable(int matches);
```

-Method to determine the amount of the win, based on the number of matches as provided. It first checks the number of spots the player played in a game. It calculates the win by taking the amount of the pay table and multiplying it by the $pergame_played$ variable. This method returns the total win.

Constructor

```
public Game();
```

-Creates the new grid of numbers in addition to generating the past 100 drawn games (that occurs in the Grid class via the super class constructor).

Implementation

```

import java.util.Random;
import java.util.Scanner;
import java.time.LocalTime;

/**
 * Class for playing a game of Keno. Will contain information for:
 * 1. The grid of numbers in the game
 * 2. The amount of numbers the player wants to play
 * 3. How many games the user wants to play
 * 4. How much they are betting per game played
 *
 * @author Alex Maureau
 */
class Game {
    //the grid of numbers
    private Grid grid;

    //user entries (or quick pick numbers)
    private int entries[];

    //needed variables
    private int nums_played = 0, games_played = 0,
                pergame_played = 0, total_wager = 0,
                game_id = 0;

    //LocalTime object, used to generate game_id
    private LocalTime time;

    /**
     * Constructor. Creates a new Grid of numbers, which also
     * invokes the inherited PastNums constructor, creating the 100
     * past games
     */
    public Game(){
        //start new grid here, which also generates 100 past games
        grid = new Grid();
    } //constructor

    //get methods
    public int getGamesPlayed() { return games_played; }
    public int getTotalWager() { return total_wager; }

    //method to detect if number was drawn in a game
    private boolean isDrawn(int n) {
        for(int i = 1; i < entries.length; i++)
            if(n == entries[i]) return true;
        return false;
    } //isDrawn

    //print the numbers the user has for a game
}

```

```

private void printEntries() {
    System.out.print("Your numbers: ");
    for(int i = 1; i <= entries.length-1; i++)
        System.out.print(entries[i] + " ");

    System.out.println();
} //printEntries

/**
 * First game of the session so gather information for the
 * 1. Amount of numbers the user wants to play (1-10)
 * 2. Whether or not the user wants a Quick Pick or to enter
 *     their own numbers
 * 3. How many consecutive games the user wants to play
 * 4. The amount wagered per game
 *
 * @param p the current Player for the game
 */
private void firstGame(Player p) {
    Scanner s = new Scanner(System.in);

    while(true) {

        System.out.print("How many numbers are you playing? (1 to
10): ");
        nums_played = s.nextInt();

        while(nums_played < 1 || nums_played > 10) {
            System.out.print("Try again please (1 to 10): ");
            nums_played = s.nextInt();
        }

        //create array based on the number choice
        //+1 on index to allow for 1 to 10
        entries = new int[nums_played+1];

        //printEntries();

        System.out.print("Quick pick? Y for Yes/N for No: ");
        char action = ' ';

        action = s.next().charAt(0);

        //invalid entry. try again
        while(action != 'Y' && action != 'y'
        && action != 'N' && action != 'n') {
            System.out.println("Invalid entry. Try again.");
            System.out.print("Y for yes/N for no: ");
            action = s.next().charAt(0);
        }
    }
}

```

```

if(action == 'Y' || action == 'y') {
    //quick pick numbers
    Random qp = new Random();

    //first number
    entries[1] = qp.nextInt(80) + 1;

    //QP up to nums times.
    //If only 1 number is chosen, loop will not run
    for(int i = 2; i <= nums_played; i++) {
        int temp = qp.nextInt(80) + 1;
        //if number is in the array already,
        //then keep drawing a new one
        while(isDrawn(temp)) {
            temp = qp.nextInt(80) + 1;
        }
        entries[i] = temp;
    }
    //desired number of entries so continue
} else if(action == 'N' || action == 'n'){
    //choosing numbers
    //first number entry:
    System.out.print("Entry 1: ");
    int temp = s.nextInt();

    while(temp < 1 || temp > 80) {
        System.out.print("Try entry 1 again: ");
        temp = s.nextInt();
    }
    entries[1] = temp;

    //if only one number is being entered
    //then loop will not run
    for(int i = 2; i <= nums_played; i++) {
        //error check entry here. otherwise
        System.out.print("Entry " + i + ": ");
        temp = s.nextInt();

        while(isDrawn(temp)
            || (temp < 1 || temp > 80)) {
            System.out.print("Try entry " + i
                + " again: ");
            temp = s.nextInt();
        }

        //otherwise good entry
        entries[i] = temp;
    }
}

//how many consecutive games does the user want to play

```

```

        System.out.print("How many games consecutive games are you
playing? (1-10): ");
        games_played = s.nextInt();

        //error check for games_played
        while(games_played <= 0 || games_played > 10) {
            //try again for consecutive games
            System.out.print("Try again please. "
+ "How many games consecutive games are you playing? "
+ "(1-10): ");
            games_played = s.nextInt();
        }

        System.out.print("How much per game? (1, 2, 5, 10 or 20): ");
        pergame_played = s.nextInt();

        //error check for pergame_played
        while(pergame_played != 1 && pergame_played != 2
            && pergame_played != 5 && pergame_played != 10
            && pergame_played != 20) {
            //try again for consecutive games
            System.out.print("Try again please. "
+ "How much per game? (1, 2, 5, 10 or 20): ");
            pergame_played = s.nextInt();
        }

        //total wager is the number of games played
        //times amount per game
        total_wager = pergame_played * games_played;

        //confirm the total wager being placed
        System.out.println("Total wager will be $" + total_wager);
        System.out.print("Confirm bet? Y for yes/N for no: ");
        action = s.next().charAt(0);

        //invalid entry. try again
        while(action != 'Y' && action != 'y'
            && action != 'N' && action != 'n') {
            System.out.println("Invalid entry. Try again.");
            System.out.print("Y for yes/N for no: ");
            action = s.next().charAt(0);
        }

        if(action == 'Y' || action == 'y') {
            //if the wager is more than the current bankroll
            //throw the exception, which will terminate game
            if(p.getBankroll() - total_wager < 0)
                throw new BankrollException("Not enough to make wager");
            else
                p.setBankroll(-total_wager);
        }
    }
}

```

```

        //break loop
        break;
    }else if(action == 'N' || action == 'n') {
        //go back to beginning of loop
        continue;
    }

} //while
} //firstGame

/**
 * Method to play the game. If played_game is true
 * then we do not need to setup a new game.
 *
 * @param played_game represents whether we played a game already
 * @param p the current Player for the game
 */
public void play(boolean played_game, Player p) {
    //playing a game to repeat numbers
    if(!played_game) {
        firstGame(p);
    }

    System.out.println("Good luck!");
    System.out.println();

    //game_id is unique based on time of day
    time = LocalTime.now();
    game_id = time.toSecondOfDay();

    System.out.println("Now playing game " + game id);
    System.out.println();

    //otherwise print entries and draw numbers
    printEntries();

    System.out.println();

    //shuffle numbers and print the grid to the console
    grid.shuffle();
    grid.printGrid(entries);
} //play

/**
 * Method to determine which numbers are hot,
 * meaning they have been the most frequently drawn
 *
 * Prints out the 10 hottest numbers
 *
 * Uses selection sort for the parallel arrays
 */

```

```

public void hotNumbers() {
    //copies of current arrays from Grid class
    int[] hot = grid.drawn_times;
    int hot_index[] = grid.drawn_index;

    System.out.println("HOT NUMBERS: ");

    //selection sort by highest to lowest
    for(int i = 1; i < hot.length-1; i++){
        int indexMax = i;
        for(int j = i+1; j < hot.length; j++){
            if(hot[j] >= hot[indexMax]) indexMax = j;
        }
        //found the highest elements index so swap here:
        if(hot[i] != hot[indexMax]){
            int temp = hot[i];
            hot[i] = hot[indexMax];
            hot[indexMax] = temp;

            int temp2 = hot_index[i];
            hot_index[i] = hot_index[indexMax];
            hot_index[indexMax] = temp2;
        }
    }

    //print the 10 hottest numbers
    for(int i = 1; i <= 10; i++)
        System.out.print(hot_index[i] + " ");
    System.out.println();
} //hotNumbers

/**
 * Method to determine which numbers are cold,
 * meaning they have been the least frequently drawn
 *
 * Prints out the 10 coldest numbers
 *
 * Uses selection sort for the parallel arrays
 */
public void coldNumbers() {
    //copies of current arrays from Grid class
    int[] cold = grid.drawn_times;
    int[] cold_index = grid.drawn_index;

    System.out.println("COLD NUMBERS: ");

    //selection sort by lowest to highest
    for(int i = 1; i < cold.length-1; i++){
        int indexMin = i;
        for(int j = i+1; j < cold.length; j++){
            if(cold[j] <= cold[indexMin]) indexMin = j;
        }
    }
}

```

```

        }
        //found the smallest elements index so swap here:
        if(cold[i] != cold[indexMin]){
            int temp = cold[i];
            cold[i] = cold[indexMin];
            cold[indexMin] = temp;

            int temp2 = cold_index[i];
            cold_index[i] = cold_index[indexMin];
            cold_index[indexMin] = temp2;
        }
    }

    //print the 10 coldest numbers
    for(int i = 1; i <= 10; i++)
        System.out.print(cold_index[i] + " ");
    System.out.println();
} //coldNumbers

/**
 * Method will check for the number of matches the Player
 * has from their game.
 *
 * @return total amount of the win for user
 */
public int checkWin() {
    //checking the entries array for the
    //number of matches the Player has
    int matches = 0;
    for(int i = 1; i < entries.length; i++) {
        for(int j = 1; j <= 20; j++) {
            //if number from player has been drawn
            if(entries[i] == grid.grid[j]) {
                matches++;
                break;
            }
        }
    }

    //get win based on pay table
    int total_win = payTable(matches);

    //print win information
    System.out.println();
    System.out.println("You matched " + matches + " numbers.");
    System.out.println("Total game wager: $" + pergame_played);
    System.out.println("Amount won: $" + total_win);

    //return the amount of the win based on number of matches
    return total_win;
} //checkWin

```

```

/**
 * Method to determine the amount of the win based on the
 * number of matches provided
 *
 * @param matches number of matches player has in their game
 * @return total amount of the win
 */
private int payTable(int matches) {
    int total_win = 0;

    //determine win based on number of games played
    //grid determined via NY Lottery payouts
    switch(nums_played) {
        case 1: //1 spot game
            if(matches == 1) {
                total_win = 2 * pergame_played;
            }
            break;

        case 2: //2 spot game
            if(matches == 2) {
                total_win = 10 * pergame_played;
            }
            break;

        case 3: //3 spot game
            if(matches == 2) {
                total_win = 2 * pergame_played;
            } else if(matches == 3) {
                total_win = 23 * pergame_played;
            }
            break;

        case 4: //4 spot game
            if(matches == 2) {
                total_win = 1 * pergame_played;
            } else if(matches == 3) {
                total_win = 5 * pergame_played;
            } else if(matches == 4) {
                total_win = 55 * pergame_played;
            }
            break;

        case 5: //5 spot game
            if(matches == 3) {
                total_win = 2 * pergame_played;
            } else if(matches == 4) {
                total_win = 20 * pergame_played;
            } else if(matches == 5) {

```

```

        total_win = 300 * pergame_played;
    }
    break;

case 6: //6 spot game
    if(matches == 3) {
        total_win = 1 * pergame_played;
    }else if(matches == 4) {
        total_win = 6 * pergame_played;
    }else if(matches == 5) {
        total_win = 55 * pergame_played;
    }else if(matches == 6) {
        total_win = 1000 * pergame_played;
    }
    break;

case 7: //7 spot game
    if(matches == 0) {
        total_win = 1 * pergame_played;
    }else if(matches == 4) {
        total_win = 2 * pergame_played;
    }else if(matches == 5) {
        total_win = 20 * pergame_played;
    }else if(matches == 6) {
        total_win = 100 * pergame_played;
    }else if(matches == 7) {
        total_win = 5000 * pergame_played;
    }
    break;

case 8: //8 spot game
    if(matches == 0) {
        total_win = 2 * pergame_played;
    }else if(matches == 5) {
        total_win = 6 * pergame_played;
    }else if(matches == 6) {
        total_win = 75 * pergame_played;
    }else if(matches == 7) {
        total_win = 550 * pergame_played;
    }else if(matches == 8) {
        total_win = 10000 * pergame_played;
    }
    break;

case 9: //9 spot game
    if(matches == 0) {
        total_win = 2 * pergame_played;
    }else if(matches == 5) {
        total_win = 5 * pergame_played;
    }else if(matches == 6) {
        total_win = 20 * pergame_played;
    }
}

```

```

        }else if(matches == 7) {
            total_win = 125 * pergame_played;
        }else if(matches == 8) {
            total_win = 3000 * pergame_played;
        }else if(matches == 9) {
            total_win = 30000 * pergame_played;
        }
        break;

    case 10: //10 spot game
        if(matches == 0) {
            total_win = 5 * pergame_played;
        }else if(matches == 5) {
            total_win = 2 * pergame_played;
        }else if(matches == 6) {
            total_win = 10 * pergame_played;
        }else if(matches == 7) {
            total_win = 45 * pergame_played;
        }else if(matches == 8) {
            total_win = 300 * pergame_played;
        }else if(matches == 9) {
            total_win = 5000 * pergame_played;
        }else if(matches == 10) {
            total_win = 100000 * pergame_played;
        }
        break;
    } //switch

    return total_win;
} //totalwin
} //Game

```

Keno.java

Main class, containing main() method for the game of Keno.

Imports

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

Instance variables

```
private static Player p;
private static Game g;
private static char action;
private static boolean played_game;
private static int total_win, total_games_played;
```

Methods (header lines)

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

-Main method for program. Will ask user for name and age, then create a new Player and Game. Program loops until user exits or an Exception is thrown.

```
public void playGame();
```

-Method to play X number of games as defined by the user in Game.java.

Constructor

[None]

Implementation

```
import java.util.Scanner;

/**
 * Keno.java
 *
 * Program will simulate a game of Keno. It will prompt
 * a user with a menu of options. If the user plays a
 * game, program creates a new Grid, in addition to generating
 * a past 100 drawn games to determine hot and cold numbers.
 *
 * @author Alex Maureau
 *
 */
public class Keno {
    //information on our Player
    private static Player p;

    //the game will be run here
    private static Game g;

    //the user entry
    private static char action;

    //have we played at least 1 game
    private static boolean played_game;

    //tracking how much we have won on a game and how
    //many total games we played
    private static int total_win, total_games_played;

    public static void main(String args[]){
        Scanner s = new Scanner(System.in);

        System.out.println("Please enter your name & bankroll:");

        //get name and bankroll from user
```

```

String name = s.next();
int bank = s.nextInt();

//initial values
action = ' ';
played_game = false;

try {
    //create a new Player with entered information
    p = new Player(name, bank);

    //create a new Game
    g = new Game();

    //initial values
    total_win = 0;
    total_games_played = 0;

    //keep playing the game until the user ends it
    while(true) {
        System.out.println("Please enter a character as seen at left.");

        //had played a game already so check if user wants to
        //repeat numbers and bets
        if(played_game) {
            System.out.println("P -> Play new game");
            System.out.println("R -> Repeat last game");
            System.out.println("E -> Exit program");

            //user entry
            action = s.next().charAt(0);

            //user decides to end game so break the loop
            if(action == 'E' || action == 'e') {
                break;
            } else if(action == 'P' || action == 'p'){
                //playing a new game so set flag to false
                played_game = false;
                playGame();
            } else if(action == 'R' || action == 'r') {
                //repeating a game. need to check current bankroll
                //versus the total ticket wager. if it will get the
                //bankroll to 0 or less, throw exception and end game
                if(p.getBankroll() - g.getTotalWager() < 0) {
                    throw new BankrollException(
                        "Error on repeat game: not enough to make repeat wager");
                } else {
                    //otherwise update player bankroll
                    p.setBankroll(-g.getTotalWager());
                }
            }
        }
    }
}

```

```

        //we are repeating a played game so set flag
        played_game = true;

        //now play the game
        playGame();
    }

} else {
    //first pass for a game
    System.out.println("P -> Play new game");
    System.out.println("E -> Exit program");

    //user entry
    action = s.next().charAt(0);

    //user decides to end game so break the loop
    if(action == 'E' || action == 'e') {
        break;
    } else if(action == 'P' || action == 'p'){
        //play first game so false flag
        played_game = false;

        //now play the game
        playGame();
    }
    //if not the right entry, the code will go back
    //to the top and ask again
}
}

} catch(Exception e) {
    //print error
    System.out.println(e);
}

//exit output
int starting = p.getStartingBankroll();
int remains = p.getBankroll();

System.out.println("Thank you for playing!");
System.out.println("You played: " + total_games_played
                  + " games");
System.out.println("Remaining bankroll: $" + remains);

if(remains - starting < 0) {
    System.out.println("Net loss: "
                      + (remains - starting));
} else if(remains - starting > 0) {
    System.out.println("Net win: "
                      + (remains - starting));
} else { //even
    System.out.println("You are even! ");
}

```

```

        }
    } //main

    /**
     * Method to play X number of games as defined in Game.java
     *
     * No arguments needed
     */
    private static void playGame() {
        //play the game here
        g.play(played_game, p);

        //increase total games played
        total_games_played++;

        //we have now played at least 1 game
        played_game = true;

        //will calculate the amount won. 0 has no effect
        //on player bankroll
        total_win = g.checkWin();

        //update bankroll with current win total. 0 has no effect
        p.setBankroll(total_win);

        //if only 1 game was selected to be played, show
        //the hot and cold numbers
        if(g.getGamesPlayed() == 1) {
            try {
                //short game break
                Thread.sleep(3000);

                if(total_games_played % 3 == 0){
                    //display both hot and cold drawn numbers
                    System.out.println();
                    g.hotNumbers();
                    g.coldNumbers();
                    System.out.println();
                }

                //sleep 3 seconds before continuing
                Thread.sleep(3000);
            }catch(Exception e) {}

        }else {
            //play rest of desired games.
            for(int i = 2; i <= g.getGamesPlayed(); i++) {
                try {
                    //short break
                    Thread.sleep(3000);

```

```

        if(total_games_played % 3 == 0){
            //display both hot and cold drawn numbers
            System.out.println();
            g.hotNumbers();
            g.coldNumbers();
            System.out.println();
        }

        //sleep 3 seconds
        Thread.sleep(3000);

        //5 second countdown clock
        System.out.print("Next game beginning in ");
        for(int j = 5; j > 0; j--) {
            System.out.print(j + "...");
            Thread.sleep(1000);
        }
        System.out.println();
    }catch(Exception e) {}

        //play the next game and increase count
        g.play(player_game, p);
        total_games_played++;

        //now check the win amount
        System.out.println();
        total_win = g.checkWin();

        //update bankroll with current win total.
        p.setBankroll(total_win);
    }
}

//display player information before deciding to
//play again or repeat game
System.out.println();
System.out.println(p);
} //playGame
} //class

```

Sample Output

Here is a short run showing a possible output to the game. The results will always vary due to Random class being used.

Please enter your name and bankroll:

Alex 100

Please enter a character as seen at left.

P -> Play new game

E -> Exit program

p

You currently have \$100

How many numbers are you playing? (1 to 10): 5

Quick pick? Y for Yes/N for No: n

Entry 1: 16

Entry 2: 1

Entry 3: 80

Entry 4: 54

Entry 5: 22

How many games consecutive games are you playing? (1-10): 4

How much per game? (1, 2, 5, 10 or 20): 10

Total wager will be \$40

Confirm bet? Y for yes/N for no: y

Good luck!

Now playing game 75737

Your numbers: 16 1 80 54 22

Numbers drawn:

20 22* 26 10 71 56 37 47 27 61 67 4 42 40 9 14 53 44 45 80*

1	2	3	--	5	6	7	8	--	--
11	12	13	--	15	16	17	18	19	--
21	--	23	24	25	--	--	28	29	30
31	32	33	34	35	36	--	38	39	--
41	--	43	--	--	46	--	48	49	50
51	52	--	54	55	--	57	58	59	60
--	62	63	64	65	66	--	68	69	70
--	72	73	74	75	76	77	78	79	--

You matched 2 numbers.

Total game wager: \$10

Amount won: \$0

Next game beginning in 5...4...3...2...1...

Good luck!

Now playing game 75768

Your numbers: 16 1 80 54 22

Numbers drawn:

57 29 3 62 40 10 28 26 19 66 39 49 41 44 17 61 70 48 51 67

1	2	--	4	5	6	7	8	9	--
11	12	13	14	15	16	--	18	--	20
21	22	23	24	25	--	27	--	--	30
31	32	33	34	35	36	37	38	--	--
--	42	43	--	45	46	47	--	--	50
--	52	53	54	55	56	--	58	59	60
--	--	63	64	65	--	--	68	69	--
71	72	73	74	75	76	77	78	79	80

You matched 0 numbers.

Total game wager: \$10

Amount won: \$0

Next game beginning in 5...4...3...2...1...

Good luck!

Now playing game 75799

Your numbers: 16 1 80 54 22

Numbers drawn:

23 75 62 1* 68 8 22* 76 65 27 54* 29 41 20 15 60 43 9 56 10

--	2	3	4	5	6	7	--	--	--
11	12	13	14	--	16	17	18	19	--
21	--	--	24	25	26	--	28	--	30
31	32	33	34	35	36	37	38	39	40
--	42	--	44	45	46	47	48	49	50
51	52	53	--	55	--	57	58	59	--
61	--	63	64	--	66	67	--	69	70
71	72	73	74	--	--	77	78	79	80

You matched 3 numbers.

Total game wager: \$10

Amount won: \$20

HOT NUMBERS:

52 41 5 43 34 8 56 24 63 44

COLD NUMBERS:

51 79 40 57 77 3 70 26 53 19

Next game beginning in 5...4...3...2...1...

Good luck!

Now playing game 75830

Your numbers: 16 1 80 54 22

Numbers drawn:

67 36 35 52 75 19 76 58 55 78 12 39 15 48 16* 42 10 5 2 70

1	--	3	4	--	6	7	8	9	--
11	--	13	14	--	--	17	18	--	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	--	--	37	38	--	40
41	--	43	44	45	46	47	--	49	50
51	--	53	54	--	56	57	--	59	60
61	62	63	64	65	66	--	68	69	--
71	72	73	74	--	--	77	--	79	80

You matched 1 numbers.

Total game wager: \$10

Amount won: \$0

Alex, you have \$80 left

Please enter a character as seen at left.

P -> Play new game

R -> Repeat last game

E -> Exit program

e

Thank you for playing!

You played: 4 games

Remaining bankroll: \$80

Net loss: -20

Well, we lost money, but hey, it was fun!

This encompassed a lot of the different aspects of Java. Feel free to modify or add additional features to the game, like a bonus number being drawn, or double payouts. Whatever you want to try!

APPENDIX D

Programming Challenges

For each question, write a Java program, to solve the question. There is more than 1 way to solve these questions. No solutions will be provided, as these are left up to you to solve.

Log on to www.cstutoringcenter.com/problems to submit your answers and climb our leader board! **Register for a FREE account to submit the answers.** (Note: Problem titles are as they appear on our website).

Beginner

Factorial Fun

Given the first few factorials below:

$$1! = 1$$

$$2! = 2 \times 1 = 2$$

$$3! = 3 \times 2 \times 1 = 6$$

$$4! = 4 \times 3 \times 2 \times 1 = 24$$

What is the sum of the first 15 factorials (not including 0!)?

Sum of Digits

As an example, the sum of the digits of 2 to the 10th power is:

$$2^{10} = 1024 \Rightarrow 1+0+2+4 \Rightarrow 7$$

What is the sum of the digits of 2^{50} ?

Reverse Alphabet Codes

Given the following information:

$$a = 26, b = 25, c = 24, d = 23 \dots x = 3, y = 2, z = 1$$

What is the sum of each letter of this sentence: "The quick brown fox jumped over the cow"? (Note, a white space has no value).

Prime Numbers I

In math, a prime number is a number only divisible by 1 and itself.

Given the first few prime numbers:

2 3 5 7 11 13 17 ...

What is the sum of the first 250 prime numbers? You are seeing the first 7 primes above.

Fifty Fifty

In honor of our 50th problem, what is the sum of the 50th Fibonacci number, 50th prime number and the 50th odd number?

Recall that the Fibonacci sequence looks like:

1 1 2 3 5 8 13

the prime sequence looks like:

2 3 5 7 11 13

and we all know the sequence of odd numbers which is trivial.

Morse Code Factorial

Morse code was an ancient way of sending messages in war time. It consists of only dots and dashes in sequence.

Below is the Morse Code for the digits 0 to 9:

0	----	6	-....
1	.----	7	--...
2	..---	8	---..
3	...--	9	----.
4-		
5		

Using this code, how many dots in total are shown in $100!$ (the actual number you get from 100 factorial)?

A Math Puzzle

Given this scheme below:

$$\text{ABCDE} / \text{FGHI} = 9$$

How many solutions are there to the puzzle for the digits 1 to 9 being used once and only once in a particular set of numbers?

12345 / 6789 VALID

44213 / 7689 NOT VALID (5 is not there)

Lojban I

Counting in Lojban, an artificial language developed over the last forty years, is easier than in most languages. The numbers from zero to nine are:

1 pa 4 vo 7 ze
2 re 5 mu 8 bi 0 no
3 ci 6 xa 9 so

Larger numbers are created by gluing the digits together. For example, 123 is pareci.

How many times does the character 'o' appear when converting each number from 1 to 10,000 (inclusive) to Lojban?

Enter the answer as a lowercase Lojban string with no whitespaces!

Intermediate

The Birthday Paradox

If there are 3 people in a room, the probability of two of those people having THE SAME birthday is 0.008%. For 4 people, it is 0.016% and for 5 people, it is 0.027%.

Using the same rule of at least two people in a room of size 25, what is the probability (rounded to the nearest 1000th) of two people having THE SAME BIRTHDAY?

For the purpose of the calendar, there are 365 days in the year (Feb. 29th is counted a Mar. 1st).

US Telephone Keypads

Given the following information about a US telephone touchtone keypad:

1: (NONE)	2: A,B,C	3: D,E,F
4: G,H,I	5: J,K,L	6: M,N,O
7: P,R,S	8: T,U,V	9: W,X,Y

Calculate the product of each characters value. As an example, say the user enters: "Practice", the product would be:

$$7 * 7 * 2 * 2 * 8 * 4 * 2 * 3 = 37,632$$

What is the value of this string: "Programming Challenges are fun"?

Sum of Primes

What is the sum of all prime numbers below 5000?

A Numerical Pattern

Given the following number representing a pattern:

$$2053 = 2^0 + 5^3 = 1 + 125 = 126$$

Look above at the number "20"; in this sequence it represents 2 to the 0 power which is 1. The other number "53" represents 5 to the 3rd power which is 125. Then final answer is calculated by the sum of each term.

Given this number:

$$342345820139586830203845861938475676$$

What is the answer using the above pattern?

Prime Numbers III

How many of the primes below 1,000,000 have the sum of their digits equal to the number of days in a fortnight?

Numerical Pattern II

Given the numerical pattern below:

$$1^2 + 2^3 + 3^4 + \dots + 18^{19} + 19^{20}$$

What is the last 5 digits of this pattern?

Not so Perfect!

In math, a perfect square is defined as the square root of a number that does not produce a remainder. The first few perfect squares are:

$$\text{sqrt}(1) = 1$$

$$\text{sqrt}(4) = 2$$

$$\text{sqrt}(9) = 3$$

$$\text{sqrt}(16) = 4$$

etc..

Using standard double precision accuracy, what is the sum of all non-perfect squares for all n ($2 \leq n \leq 10000$)?

A Prime Fibber

Jack has a stack of index cards with the first 25 prime numbers. John has a stack of index cards with the first 25 Fibonacci numbers.

What is the probability that, when they both choose a number, that Jacks number is a sub string of Johns number. As an example below:

Jack chooses --> 5

John chooses --> 233

NO

Jack chooses --> 59

John chooses --> 1597

YES

Enter the number in the form 0.XXXX where X is the digits of the answer.

Triangle Numbers

A triangle number is computed with the following formula:

$$T(n) = n(n+1)/2$$

Using the above formula, the first 5 triangle numbers are:

1 3 6 10 15 ...

What then is the sum of the digits of the 123,456,789th triangle number?

Binary Clock

A binary clock is a clock which displays sexagesimal time (military format) in a binary format. There are also two kinds of representations for a binary clock; vertical or horizontal.

Below is the horizontal representation of the time 10:37:49

(H)	0 0 1 0 1 0	(10)
(M)	1 0 0 1 0 1	(37)
(S)	1 1 0 0 0 1	(49)

so when it is grouped together as a binary string, it looks like this:

001010 100101 110001

Find the total number of 1s in the representations of military time for a full 24 hour period which elapses from 00:00:00 to 23:59:59.

How many total 1s appear for the entire year (a regular 365 day year)?

Triangle of Primes

Given the Triangle of primes.

2
3 5
7 11 13
17 19 23 29
31 37 41 43 47
53 59 61 67 71 73
....

The sum of the 6th row is 384. Give the sum of the 100th row.

N-Squared Sequence

Given that the first 2 terms of a sequence are 1, you are to take the sum of ALL previous terms and square it to generate the next term in sequence. This repeats indefinitely.

The sequence will look like this:

1	1	4	36	...	actual sequence of squared terms
1	1	2	6	actual terms

What is the sum of the digits of the 10th term in the above sequence?

Lojban II

Counting in Lojban, an artificial language developed over the last forty years, is easier than in most languages. The numbers from zero to nine are:

1 pa 4 vo 7 ze
2 re 5 mu 8 bi 0 no
3 ci 6 xa 9 so

A self-descriptive number is equal to the sum of the positions of its letters in the alphabet. For example, pareci is not self-descriptive, since 123 is not equal to $16 + 1 + 18 + 5 + 3 + 9$ (=52).

What is the product of all self-descriptive numbers below 1,000,000?

Duck Duck Goose

A group of friends are standing in a circle, playing a game. They proceed around the circle, chanting a rhyme and pointing at the next person (clockwise) on each word. When the rhyme finishes whoever is being pointed at leaves the circle. They then start again, pointing where they left off, and the game continues until only one person is left. He/she is declared the winner.

For example, suppose there are six friends (numbered 1 to 6) standing around the circle and the rhyme is "Eenee, Meenee, Mainee, Mo!" They begin with player number 1. The first person to leave the circle is number 4, followed by 2, then 1, 3 and 6. Number 5 is left (draw it out to see it first hand).

Using this pattern, what number player, beginning with player 1, is the last person standing in a game played with 100,000 players?

Advanced

Fibonacci Sequence II

Given the first few numbers of the Fibonacci sequence:

1 1 2 3 5 8 13 21 ...

What is the first Fibonacci number to contain 15 digits? Give the actual 15 digit number for the answer.

Fibonacci Sequence III

Given the first few numbers of the Fibonacci sequence:

1 1 2 3 5 8 13 21 ...

What is the sum of the first 75 Fibonacci numbers? As an example, the sum of the first 7 numbers above is 33.

A Pandigital Exponent

A pandigital number makes use of the digits 1 through 9 at least once, but not in any particular order. For example:

146523798 is a 9 digit pandigital

What is the smallest value of k such that 2^k has its last 9 digits pandigital from 1 to 9? (Note: \wedge means exponent).

This requires a more clever method to solve as the numbers will get large fast.

Pascal's Modded Triangle

By the famous binomial theorem, we can develop Pascal's Triangle. Here is the first 5 rows of the famous triangle:

1
1 1
1 2 1

1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
.....

A few things to observe; the first and last numbers of each row is 1. To calculate the other numbers, it is simply the sum of the above numbers to the left and right.

Let's just say that we wanted to keep it short and sweet. For the sake of being lazy, we will mod the triangle by 100 to only observe numbers no larger than 99 in the triangle. The triangle above is still the same even when modding it by 100. The reason for the mod is because the numbers get large really fast!

What is the sum of the sums of the first 25,000 rows of Pascal's "Modded" Triangle?

Hexagonal Numbers

A hexagonal number is computed as follows using the formula:

$$H(n) = n(2n-1)$$

As shown, the first 5 hexagonal numbers are:

1 6 15 28 45

How many of the first 100,000,000 hexagonal numbers are evenly divisible by all the numbers from 1 through 20?

The 20th Century

You are given the following pieces of information about the 20th century:

- Jan. 1st 1900 was a Monday
- 30 days has September, April, June and November. All the rest have 31 except February which has 28 and 29 on a leap year.
- A leap year occurs on every year divisible by 4 except on a century unless it is divisible by 400.

Using your knowledge and the info above, how many Fridays during the 20th century were divisible by 3 or 5? (1 Jan 1901 to 31 Dec 2000).

Non Decreasing Digits

A number is said to be made up of non-decreasing digits if all the digits to the left of any digit are less than or equal to that digit. For example, the four-digit number 1234 is composed of digits that are non-decreasing. Some other four-digit numbers that are composed of non-decreasing digits are 0011, 1111, 1112, 1122, 2223. As it turns out, there are exactly 715 four-digit numbers composed of non-decreasing digits.

Notice that leading zeroes are required: 0000, 0001, 0002 are all valid four-digit numbers with nondecreasing digits.

How many 25 digit numbers are composed of non-decreasing digits?

Best of luck solving these questions! Remember, register for free to submit the answers and climb the board. Much more problems are available online as new challenges are added frequently.

APPENDIX E

Jeopardy! Fun! How Much Can You Win on the Game of Jeopardy!

The long-running game show Jeopardy! was created by Merv Griffin back in 1964. It has had a slew of hosts, most notably, and currently Alex Trebek, who has hosted the show since 1984.

RULES & GAMEPLAY

The game itself consists of three players (one returning champion and two others). The game also consists of a single Jeopardy round, a double Jeopardy round, and a final Jeopardy round, played in that order.

Game Boards

The game board consists of 6 categories, each of which has 5 clues. At the present time (2019), the current game board looks like this for the single Jeopardy round:

\$200	\$200	\$200	\$200	\$200	\$200
\$400	\$400	\$400	\$400	\$400	\$400
\$600	\$600	\$600	\$600	\$600	\$600
\$800	\$800	\$800	\$800	\$800	\$800
\$1000	\$1000	\$1000	\$1000	\$1000	\$1000

The board for the double Jeopardy round consists of the above amounts doubled. There are still 6 categories and 5 clues in each one.

For further background, the clues themselves are answers. A contestant must give their response in question form. For this model, this is irrelevant, since we will not be dealing with actual responses from a contestant.

Daily Double(s)

A daily double will allow a contestant to potentially double their current winnings. There is one daily double in single Jeopardy & two in double Jeopardy. These can

appear in ANY of the places above in either round. That does include the first row (which many people think can never occur).

What this will do is enable a player to wager any amount of or all of their money (with a minimum of a \$5 bet). If the current bank roll of the contestant is negative or \$0, the contestant may wager up to the highest value clue on the board (\$1000 in single Jeopardy and \$2000 in double Jeopardy).

Single Jeopardy

The first round of game play. Here, the returning champion picks the first clue from the game board. The round continues until all clues are completed OR time has expired. This round also consists of one (1) daily double.

Double Jeopardy

The second round of game play. Here, the contestant with the lowest bank roll picks the first clue from the game board. The round continues until all clues are completed OR time has expired. This round also consists of two (2) daily doubles.

At the end of this round, any or all players with a \$0 or negative balance will NOT play in final Jeopardy (even if that means all of the players).

Final Jeopardy

The final round of game play. Here, one (1) clue is asked to the remaining contestants. They must decide how much they want to wager BEFORE the clue appears. They can wager up to their current bankroll or \$0.

The player with the highest score at the end of Final Jeopardy will be declared the "champion" and return for the next show.

HOW MUCH CAN YOU WIN?

As mentioned earlier, we will be assuming and/or realizing the following things about the game play:

1. There is 1 Daily Double for round 1 (single Jeopardy) and 2 in round 2 (double Jeopardy).

2. A daily double (DD) can be placed in any of the spots on the board in any round.
3. We do not care about responses, as we are talking about one contestant answering all the questions.
4. The contestant ALWAYS answers the clue correctly.

For the purpose of the model, we are interested in finding the minimum of the maximum and the true maximum values that can occur in each round of game play.

1964 Show

Let's begin with a trip down memory lane and visit the show when it first came on air in 1964. The game board for single Jeopardy was structured as such:

\$10	\$10	\$10	\$10	\$10	\$10
\$20	\$20	\$20	\$20	\$20	\$20
\$30	\$30	\$30	\$30	\$30	\$30
\$40	\$40	\$40	\$40	\$40	\$40
\$50	\$50	\$50	\$50	\$50	\$50

Round 1 "Single Jeopardy"

Highest Amount: *The DD is placed in the first row (\$10 space) in any of the categories.*

The player has answered all the questions correctly and then finds the DD to double his winnings. Simply add the amounts on the board and then double it:

$$(\$50 \bullet 6) + (\$40 \bullet 6) + (\$30 \bullet 6) + (\$20 \bullet 6) + (\$10 \bullet 5) = ? \\ (\$300) + (\$240) + (\$180) + (\$120) + (\$50) = \$890 \bullet 2 = \$1,780$$

Lowest Amount: *The DD is placed in the fifth row (\$50 space) in any of the categories.*

If the player hasn't answered any questions yet and chooses a DD first, he would have nothing to bet. Therefore, he can only bet from the minimum of \$5 or up to \$50. Add the remaining amounts on the board and add \$5:

$$(\$50 \bullet 5) + (\$40 \bullet 6) + (\$30 \bullet 6) + (\$20 \bullet 6) + (\$10 \bullet 6) + \$5 = ?$$

$$(\$250) + (\$240) + (\$180) + (\$120) + (\$60) + \$5 = \$855$$

Single Jeopardy Round Summary:

Lowest: \$855

Highest: \$1,780

Round 2 “Double Jeopardy”

The game board doubles from what it was in the first round.

\$20	\$20	\$20	\$20	\$20	\$20
\$40	\$60	\$40	\$40	\$40	\$40
\$60	\$60	\$60	\$60	\$60	\$60
\$80	\$80	\$80	\$80	\$80	\$80
\$100	\$100	\$100	\$100	\$100	\$100

Here, there are 15 possible combinations of the placements of the Daily Doubles. The bolded combinations are the only ones relevant to this model. The numbers represent the row and column, respectively:

(1,1), (1,2), (1,3), (1,4), (1,5), (2, 2), (2,3), (2,4), (2,5), (3,3), (3,4), (3,5), (4,4), (4,5), **(5,5)**

Highest amount: The two DDs are placed in the first row (\$20 space) in any two of the categories.

--The player has already won \$1,780--

If the player finds the two DDs last, add the remaining amounts on the board first:

$$(\$100 \bullet 6) + (\$80 \bullet 6) + (\$60 \bullet 6) + (\$40 \bullet 6) + (\$20 \bullet 4)$$

$$= (\$600) + (\$480) + (\$360) + (\$240) + (\$80)$$

$$= \$1,760$$

Then, add the amount won from the previous round (being \$1,780) to that total:

$$\$1,760 + \$1,780 = \$3,540$$

Then, take that amount and multiply it by 4 (doubling it twice):

$$\$3,540 \bullet 4 = \$14,160$$

Lowest amount: The two DDs are placed in the fifth row (\$100 space) in any two of the categories.

--The player has already won \$855--

If the player finds the two DDs first, take the lowest amount of money from round one (being \$855), add \$10 (representing two, \$5 DDs) and then add it to the rest of the amounts on the board:

$$\$855 + \$10 = \$865$$

$$\begin{aligned} & (\$100 \bullet 4) + (\$80 \bullet 6) + (\$60 \bullet 6) + (\$40 \bullet 6) + (\$20 \bullet 6) + \$865 \\ &= (\$400) + (\$480) + (\$360) + (\$240) + (\$120) + \$865 \\ &= \$1,600 + \$865 = \$2,465 \end{aligned}$$

Double Jeopardy Round Summary:

Lowest: \$2,465

Highest: \$14,160

Final Jeopardy

Simply just double the above game ranges to get the total for the entire game:

$$\$2,465 \bullet 2 = \$4,930$$

$$\$14,160 \bullet 2 = \$28,320$$

Therefore, the theoretical amount of money that can be won on any particular show is anywhere in the range of \$4,930 to \$28,320. Not too shabby for 1964.

Present Day Show

The game board was changed over time as the show progressed through the years. Below is current game board as of 2019:

\$200	\$200	\$200	\$200	\$200	\$200
\$400	\$400	\$400	\$400	\$400	\$400
\$600	\$600	\$600	\$600	\$600	\$600
\$800	\$800	\$800	\$800	\$800	\$800
\$1000	\$1000	\$1000	\$1000	\$1000	\$1000

Round 1 “Single Jeopardy”

Highest Amount: The DD is placed in the first row (\$200 space) in any of the categories.

The player has answered all the questions correctly and then finds the DD to double their winnings.

Add the amounts on the board and double it:

$$\begin{aligned}
 & (\$1,000 \bullet 6) + (\$800 \bullet 6) + (\$600 \bullet 6) + (\$400 \bullet 6) + (\$200 \bullet 5) \\
 & = (\$6,000) + (\$4,800) + (\$3,600) + (\$2,400) + (\$1,000) \\
 & = \$17,800 \bullet 2 = \$35,600
 \end{aligned}$$

Lowest Amount: The DD is placed in the fifth row (\$1,000 space) in any of the categories.

If the player hasn't answered any questions yet and chooses a DD first, they would have nothing to bet. Therefore, he can only bet from the minimum of \$5 or up to \$1000. Add the remaining amounts on the board and add \$5:

$$\begin{aligned}
 & (\$1,000 \bullet 5) + (\$800 \bullet 6) + (\$600 \bullet 6) + (\$400 \bullet 6) + (\$200 \bullet 6) + \$5 \\
 & = (\$5,000) + (\$4,800) + (\$3,600) + (\$2,400) + (\$1,200) + \$5 \\
 & = \$17,005
 \end{aligned}$$

Single Jeopardy Round Summary:

Lowest: \$17,005

Highest: \$35,600

Round 2 “Double Jeopardy”

The game board doubles from the previous round. Here is the game board used in double Jeopardy:

\$400	\$400	\$400	\$400	\$400	\$400
\$800	\$800	\$800	\$800	\$800	\$800
\$1200	\$1200	\$1200	\$1200	\$1200	\$1200
\$1600	\$1600	\$1600	\$1600	\$1600	\$1600
\$2000	\$2000	\$2000	\$2000	\$2000	\$2000

As noted before, there are 15 possible combinations of the placements of the Daily Doubles. The bolded combinations are the only ones relevant to this model.

(1,1), (1,2), (1,3), (1,4), (1,5), (2, 2), (2,3), (2,4), (2,5), (3,3), (3,4), (3,5), (4,4), (4,5), (5,5)

Highest amount: The two DDs are placed in the first row (\$400 space) in any two of the categories

--The player has already won \$35,600--

If the player finds the two DDs last, add the reaming amounts on the board first:

$$\begin{aligned} & (\$2,000 \bullet 6) + (\$1,600 \bullet 6) + (\$1,200 \bullet 6) + (\$800 \bullet 6) + (\$400 \bullet 4) \\ & = (\$12,000) + (\$9,600) + (\$7,200) + (\$4,800) + (\$1,600) \\ & = \$35,200 \end{aligned}$$

Then, add the amount won from the previous round (being \$35,600) to that total:

$$\$35,200 + \$35,600 = \$70,800$$

Then, take that amount and multiply it by 4 (doubling it twice):

$$\$70,800 \bullet 4 = \$283,200$$

Lowest amount: The two DDs are placed in the fifth row (\$2,000 space) in any two of the categories.

--The player has already won \$17,005--

If the player finds the two DDs first, take the lowest amount of money from round one (being \$17,005), add \$10 (representing two, \$5 DDs) and then add that total to the rest of the amounts on the board:

$$\$17,005 + \$10 = \$17,015$$

$$\begin{aligned} & (\$2,000 \bullet 4) + (\$1,600 \bullet 6) + (\$1,200 \bullet 6) + (\$800 \bullet 6) + (\$400 \bullet 6) + \$17,015 \\ &= (\$8,000) + (\$9,600) + (\$7,200) + (\$4,800) + (\$2,400) + \$17,015 \\ &= \$49,015 \end{aligned}$$

Double Jeopardy Round Summary:

Lowest: \$49,015

Highest: \$283,200

Final Jeopardy

Simply just double the final game ranges to get the total for the entire game:

$$\$49,015 \bullet 2 = \$98,030$$

$$\$283,200 \bullet 2 = \$566,400$$

Therefore, the theoretical amount of money that can be won is anywhere in the range of \$98,030 to \$566,400! Still, not too bad for the present day.

Number Play

For those that don't know, Ken Jennings appeared on the show for a record 74 consecutive appearances. Imagine him winning this present day amount everyday that he was on the show. Multiply the range by 74!

$$\$98,030 \bullet 74 = \$7,254,220$$

$$\$566,400 \bullet 74 = \$41,913,600$$

Theoretically, he could have won between \$7,254,220 & \$41,913,600! Talk about a record!

PROGRAM FOR THEORETICAL JEOPARDY

The below program simulates the theoretical path on Jeopardy as seen above, where a player is answering all the questions correctly and wagering everything on a Daily Double. A sample output is provided.

```
//Jeopardy.java
import java.util.Random;
public class Jeopardy{
    private int board[][][], sumRd1, sumRd2, row, col;

    //constructor
    public Jeopardy() {
        board = new int[5][6];
        row = col = 0;
        sumRd1 = sumRd2 = 0;
    }

    //get methods
    int getSumRd1() { return sumRd1; }
    int getSumRd2() { return sumRd2; }

    //set methods
    void setSumRd1(int s) { sumRd1 = s; }
    void setSumRd2(int s) { sumRd2 = s; }

    //method to be used for simulating the gameplay round
    //either single (round 1) or double (round 2) Jeopardy
    void questions(int round) {
        Random r = new Random();

        //thirty questions in the round
        for(int q = 1; q <= 30; q++) {
            //randomly move to a row and column
            row = r.nextInt(5);
            col = r.nextInt(6);

            //an empty space as the question has already been
            //answered
            while(board[row][col] == 0) {
                row = r.nextInt(5);
                col = r.nextInt(6);
            }
        }
    }
}
```

```

//DD found
if(board[row][col] == 2) {
    //If DD selected first, the player has
    //no money, add $5 or $1,000 (randomly picked):
    if(sumRd1 == 0) {

        //randomly choosing amount
        //(0 = $5 & 1 = $1,000)
        int amt = r.nextInt(2);

        if(amt == 0) //$/5 chosen
            sumRd1 += 5;
        else //$/1,000 chosen
            sumRd1 += 1000;

    }else{

        //Assuming player answers right & doubles
        //the money:
        if(round == 1) sumRd1 *= 2;
        if(round == 2) sumRd2 *= 2;

    }

}else{

    //Current value of the questions:
    if(round == 1) sumRd1 += (200 * (row+1));
    if(round == 2) sumRd2 += (400 * (row+1));

}

//Empty space in array to show question was asked.
board[row][col] = 0;
}
} //questions

//method to randomly place the Daily Doubles on the game board
void dailyDoubles(int round) {
    Random r = new Random();
    int ddcol1 = -1;
    for(int i = 0; i < round; i++) {
        int ddrow = r.nextInt(2);
        int ddcol = r.nextInt(6);

        //second round cannot have a DD in
        //the same column so assign location
        //for first DD to variable
        if(i == 0 && round == 2) ddcol1 = ddcol;

        //indicating row 5 (bottom)
    }
}

```

```

        if(ddrow == 1) ddrow = 4;

        //the DD has been found in that spot or the
        //column generated is the same as the first DD
        //so keep generating until it's fine
        while(board[ddrow][ddcol] == 2
            || (round == 2 && i == 1 && ddcol == ddcol1)){
            ddrow = r.nextInt(2);
            ddcol = r.nextInt(6);

            if(ddrow == 1) ddrow = 4;
        } //while

        //place the DD on the game board
        board[ddrow][ddcol] = 2;
    }
} //dailyDoubles

//method to reset the game board to all 1s before round 2
void clear() {
    //reset the game board to all 1s
    for(int r = 0; r < 5; r++)
        for(int c = 0; c < 6; c++)
            board[r][c] = 1;
} //clear
} //class

//JeopardyTest.java
import java.util.Scanner;
public class JeopardyTest{
    public static void main(String args[]) {
        Scanner s = new Scanner(System.in);
        int games = 0, score = 0;

        //prompt user for entry
        System.out.println("This is JEOPARDY! Please enter # of "
            + "games to play (between 1 - 100): ");
        games = s.nextInt();

        //error checking
        if(games > 100) games = 100;
        if(games < 1) games = 1;

        //start new game
        Jeopardy game = new Jeopardy();

        //loop to run for the amount of games desired
        for(int i = 0; i < games; i++){
            System.out.println("Game #" + (i+1));

            //play round 1

```

```

        game.clear();
        game.dailyDoubles(1);
        game.questions(1);

        score = game.getSumRd1();

        System.out.println("Round 1 end score: " + score);

        //play round 2
        game.clear();
        game.dailyDoubles(2);

        //Start round 2 the amount won from round 1:
        game.setSumRd2(score);
        game.questions(2);

        score = game.getSumRd2();

        System.out.println("Round 2 end score: " + score);

        //Final Jeopardy. Just double the game score:
        score *= 2;

        System.out.println("Final Jeopardy result: "
                           + score);
        System.out.println();

        //reset the variables for the next game:
        game.setSumRd1(0);
        game.setSumRd2(0);
    }
} //main
} //class

```

A sample run of the above program can be:

This is JEOPARDY! Please enter the # of games to play (between 1 - 100):

5

Game #1

Round 1 end score: 32800

Round 2 end score: 212800

Final Jeopardy result: 425600

Game #2

Round 1 end score: 22800

Round 2 end score: 195200

Final Jeopardy result: 390400

Game #3

Round 1 end score: 20200

Round 2 end score: 170800

Final Jeopardy result: 341600

Game #4

Round 1 end score: 33000

Round 2 end score: 210000

Final Jeopardy result: 420000

Game #5

Round 1 end score: 26800

Round 2 end score: 162000

Final Jeopardy result: 324000

EXERCISE SOLUTIONS

Solutions to Chapter 2 Exercises

Variable Exercises

Problem 1:

- a) double num;
- b) float flt;
- c) short s1, s2;
- d) int x1, y1, x2, y2;
- e) long int zzz;
- f) char AA, BB, CC;
- g) float F1, F2;
- h) boolean isPrime;
- i) String last_name;
- j) short shorty;

Problem 2:

Solutions will vary, but here is ours as an example:

- a) double num = 0.0;
- b) float flt = 5;
- c) short s1 = 2, s2 = 5;
- d) int x1 = 0, y1 = 1, x2 = 2, y2 = 4;
- e) long int zzz = 1000001;
- f) char AA = 'A', BB = 'B', CC = 'C';
- g) float F1=0.0, F2=0.0;
- h) boolean isPrime = false;
- i) String last_name = "Maureau";
- j) short shorty = 0;

Operator Exercises

Problem 1:

- a) Logical And: All parts of an expression must be true to make the entire statement true.
- b) Logical Or: One or more sides of the expression must be true for the entire statement to be true.
- c) Assignment Operator: Assigns a value to a variable.
- d) Plus Equals: Adds a given value to a variable.
- e) Modular Division: Performs modular division of two integers.
- f) Increment operator: Adds 1 to a variable. Can be used as a prefix or postfix increment.

- g) Modular Equals: Performs modular division of two numbers and assign that value to a variable.
- h) Multiply: Performs multiplication in an expression.
- i) Strictly less than: Returns true if the left side is strictly less than the right side of an expression.
- j) Greater than or equal to: Returns true if the left side of the expression is greater than or equal to the right side of an expression.
- k) Logical Equals: Returns true if both sides of the expression contain the same value and false otherwise.

Problem 2:

- b) 0
- c) 4
- d) 0
- e) 7
- f) 0
- g) 0
- h) 0
- i) 4

Problem 3:

- a) true
- b) false
- c) true
- d) true
- e) true
- f) true
- g) true
- h) false

Casting Exercises

Problem 1:

- a) 99 will be the output, since it is still in range of the short data type.
- b) 55 will be the output, since the int data type does not deal with decimal places.
- c) 9999 will be the output, since when declaring the long variable n, it will ignore all the decimal places.
- d) 501 will be the output, since the int cast of the decimal value is 2.

Problem 2:

- a) correct
- b) correct
- c) correct
- d) overflow
- e) overflow
- f) correct
- g) correct
- h) correct
- i) overflow
- j) overflow

Solutions to Chapter 5 Exercises

Problem 1:

- a) Yes!
- b) Doesn't end in 5.
- c) Nope.
- d) Yes!
- e) Bigger.
- f) Yes!
- g)
 - 1. Please enter x and y: 5 8
8
 - 2. Please enter x and y: 10 6
6
- h)
 - 1. Please enter x and y: 15 12
-2
 - 2. Please enter x and y: 12 16
5
- i)
 - 1. Please enter x and y: 11 11
2
 - 2. Please enter x and y: 20 26
2
- j)
 - 1. Please enter x and y: 5 5
Odd Odd

2. Please enter x and y: 3 4
Odd Even

Problem 2:

You took 1 tries to get it right!

The secret number is: 104

Problem 3:

a)

```
int count = 1;
while (count < 50) {
    System.out.println(count);
    count+=2;
}
```

b)

```
int count = 1;
while (count <= 10) {
    System.out.println('^');
    count++;
}
```

c)

```
int count = 2, evens = 0, sum = 0;
while (evens < 100) {
    sum += count;
    count+=2;
    evens++;
}
```

d)

```
int count = 5, total = 0;
while (total < 20) {
    System.out.println(count);
    count += 10;
    total++;
}
```

e)

```
int count = 0;
while (count < 5) {
    System.out.println("#####");
    count++;
}
```

f)

```
int r = 1, c = 1;
while (r <= 3) {
    c=1;
```

```
    while(c <= 2) {  
        System.out.print( r*c + " ");  
        c++;  
    }  
    System.out.println();  
    r++;  
}
```

Problem 4:

- a) 0123456789
- b) 6
- c) 2468101214
- d) 876543210
- e) $40 * 36 * 32 * 28 * 24 * 20 *$
- f) 01234567
- g) 123
- j) 0-
- k) 70503010
- l)
G: 9.8
G: 9.7
G: 9.6
G: 9.5
G: 9.4
G: 9.3
G: 9.2
G: 9.1
G: 9
G: 8.9
G: 8.8
G: 8.7
- m) 1314

Solutions to Chapter 6 Exercises

Problem 1:

- a) Returns an all uppercase version of the String
- b) Returns an all lowercase version of the String
- c) Removes any leading and trailing whitespace from the String

- d) Returns the character at a given index in the String
- e) Returns the index of the first occurrence of a character or String within a String.
- f) Returns a boolean value indicating whether two String objects are the same.
- g) Returns a boolean value indicating whether two String objects are the same, but in checking, it ignores the case of the Strings being compared.
- h) Returns a portion of a String beginning at a given index.
- i) Compares two String objects lexicographically, returning a negative value if the primary String comes before the argument String; 0 if they are exactly equal; and a positive value if the primary String comes after the argument String.
- j) Compares two String objects lexicographically, but ignores the case of each String. Returns the same values as compareTo().

Problem 2:

- a) This method returns a boolean value that represents if there are any tokens left in the line you are tokenizing.
- b) This method will get you the next piece of data in the String.
- c) This method returns an integer representing the number of tokens in the String.

Problem 3:

- a) PROFESSOR
- b) professor
- c) e
- d) r
- e) 5
- f) 1
- g) fessor
- h) [white space]

Problem 4:

- a) REVIEWING JAVA
- b) reviewing java
- c) N
- d) [white space]
- e) -1
- f) 10
- g) e
- h) Java

Problem 5:

Apples
ORANGES
3
rang
false

Problem 6:

CollegeSchool
School---College
-1
hool
true

Problem 7:

4
JUMP
h
H
ALEX

Problem 8:

5
abc
c
123
3
DEF
F
456+GHI
6+GHI
789
9

Problem9:
COFFEE

Problem 10:
coffee
COffEE

Solutions to Chapter 7 Exercises

Problem 1:

a) 03691215

b) 43214

c) 01

d) $15 * 13 * 11 * 9 * 7 * 5 * 3 *$

e) -6-5-4-3-6-5

f)

* - *

* - * * - *

* - * * - * * - *

* - * * - * * - * * - *

* - * * - * * - * * - * - *

g)

*555*****

h)

D: 1 R: 100

D: 2 R: 100

D: 3 R: 100

D: 4 R: 100

D: 5 R: 100

D: 6 R: 100

D: 7 R: 100

D: 8 R: 100

D: 9 R: 100

D: 10 R: 100

i) 0.25 0.5 0.75 1 1.25 1.5 1.75 2 2.25 2.5 2.75

m) $4 ** 4.25 ** 4.5 ** 4.75 ** 5 ** 5.25 ** 5.5 ** 5.75 ** 6 ** 6.25 ** 6.5 ** 6.75$

n)

25

30

35

40

45

50

o)

olleH

p)
o--eH

q)
S_nS_nS_n

r)
SSSSSS

Problem 2:

a)

```
for(int i = 1; i < 50; i+=2)
    System.out.println(i);
```

b)

```
for(int i = 0; i < 10; i++)
    System.out.println('^');
```

c)

```
int sum = 0;
for(int i = 2, j=0; j < 100; i+=2, j++)
    sum += i;
```

d)

```
for(int i = 5, j=0; j < 20; i+=10, j++)
    System.out.println(i);
```

e)

```
for(int i = 0; i < 5; i++)
    System.out.println("#####");
```

f)

```
for(int i = 1; i <= 3; i++) {
    for(int j = 1; j <= 2; j++)
        System.out.print(i*j + " ");
System.out.println();
}
```

Problem 3:

a)
1 odd
2 3 odd
4 5 odd
6 9 odd

b)
infinite

c)
infinite

d)
0 1 2 3 4 5 6 0 1 2 3 4 5 0 1 2 3 4 0 1 2 3

e)
10
9
8
7
6
4
2

f)
1
1
22
1
22
333
1
22
333
4444
1
22
333
4444
55555

g)
++++++

Problem 4:

```
public class Problem4{
    public static void main(String args[]){
        int n = 0;
        n = Integer.parseInt(args[0]);

        if(n <= 0)
            System.exit(1);
        for(int i = 0; i < n; i++){
            for(int j = 0; j <= i; j++){
                System.out.print((n-j));
            }
            System.out.println();
        }
    } //main
} //class
```

Problem 5:

```
public class Problem5{
    public static void main(String args[]){
        int n = 0;
        n = Integer.parseInt(args[0]);

        if(n <= 0)
            System.exit(1);

        for(int i = 0; i < n; i++){
            for(int j = 0; j < n; j++){
                if(j < n-i-1) System.out.print(" ");
                else System.out.print((j+1));
            }
            System.out.println();
        }
    } //main
} //class
```

Problem 6:

```
public class Problem6{
    public static void main(String args[]){
        int n = 0;
        n = Integer.parseInt(args[0]);

        if(n <= 0)
            System.exit(1);

        for(int i = 1; i <= n; i++){
            for(int j = 1; j <= n; j++){
                if(i == 1 || i == n || i % 2 == 1
                   || j == 1 || j == n || j % 2 == 1)
                    System.out.print("*");
                else
                    System.out.print(" ");
            }
            System.out.println();
        }
    } //main
} //class
```

Problem 7:

```
public class Problem7{
    public static void main(String args[]){
        int n = 0;
        n = Integer.parseInt(args[0]);

        if(n <= 0 || n%2 == 0)
            System.exit(1);
```

```

        for(int i = 1; i <= n; i++) {
            for(int j = 1; j <= n; j++) {
                if(j == i || j == n-i+1) System.out.print("X");
                else System.out.print(" ");
            }
            System.out.println();
        }
    } //main
} //class

```

Problem 8:

```

public class Problem8{
    public static void main(String args[]){
        int n = 0;
        n = Integer.parseInt(args[0]);

        if(n <= 0 || n%2 == 0)
            System.exit(1);

        for(int i = 1; i <= n; i++){
            for(int j = 1; j <= n; j++){
                if(i == 1 || i == n || j == n-i+1)
                    System.out.print("X");
                else
                    System.out.print(" ");
            }
            System.out.println();
        }
    } //main
} //class

```

Output Tracing

Problem 9:

The output is:

0
-3
-6
-9
-12
-15
-18
-21
-24
-27
-30

-33

DONE

Problem 10:

1. The output is:

Dumb!

Even!

Problem 11:

1. The output is:

Dumb! 5

Odd!

Problem 12:

1. The output is:

Dumb + Dumber

Zero is the hero!

Problem 13:

1. The output is:

61

60

59

58

57

56

Problem 14:

1. The output is:

- a. Given args[0] = 6 and args[1] = 9:
Whoops!

- b. Given args[0] = 6 and args[1] = 10:

16

17

18

19

20

- c. Given args[0] = 10 and args[1] = 6:
{no output shown}

Problem 15:

1. The output is:

4
6
8
10

Solutions to Chapter 8 Exercises

Problem 1:

- a) public static int sum(int a, int b, int c);
- b) public static void printChar(char c, int n);
- c) public static void triangle(int n);
- d) public static void perfectSq(int n);
- e) public static boolean isLeapYear(int year);
- f) public static boolean isOdd(int n);
- g) public static boolean isEven(int n);
- h) public static boolean isEvenDiv(int n);
- i) public static int evenSum(int a, int b);
- j) public static int oddSum(int a, int b);
- k) public static String reverseStr(String s);
- l) public static String toLower(String s);
- m) public static char getSub(String s, int n);
- n) public static double inBetween(double d1, double d2);
- o) public static boolean isAllEven(int n);
- p) public static boolean isAlpha(String s);
- q) public static boolean isAlphaNumeric(String s);
- r) public static void dupe8(int n);
- s) public static int drop8(int n);
- t) public static void squareNum(int n);
- u) public static double frac(int n1, int n2);
- v) public static int inString(String s, char c);
- w) public static int stringInString(String s, String t);

Problem 2:

- a)
public static int sum(int a, int b, int c){ return a+b+c; }
- b)
public static void printChar(char c, int n){
 for(int i = 0; i < n; i++) System.out.print(c);
 System.out.print("\n");
}

```

c)
public static void triangle(int n){
    for(int r = 0; r < n; r++){
        for(int c = 0; c < r; c++){
            System.out.print("*");
        }
        System.out.print("\n");
    }
}

d)
public static void perfectSq(int n){
    for(int i = 1; i <= n; i++){
        System.out.print(i*i + " ");
    }
}

e)
public static boolean isLeapYear(int year){
    if(year % 4 == 0){
        if(year % 100 != 0 && year % 400 == 0){
            return true;
        }else
            return false;
    }
    return false;
}

f)
public static boolean isOdd(int n){ return n%2==1; }

g)
public static boolean isEven(int n){ return n%2==0; }

h)
public static boolean isEvenDiv(int n){ return(n%2==0 && n%5==0); }

i)
public static int evenSum(int a, int b){
    int sum = 0;
    for(int i = a; i <= b; i++)
        if(i % 2 == 0) sum += a;
    return sum;
}

j)
public static int oddSum(int a, int b){
    int sum = 0;
    for(int i = a; i <= b; i++)
        if(i % 2 == 1 || i % 5 == 0) sum += a;
    return sum;
}

```

```
}

k)
public static String reverseStr(String s) {
    String str = "";

    for(int i = s.length()-1; i >= 0; i--) {
        str += s.charAt(i);
    }
    return str;
}

l)
public static String toLower(String s) {
    return s.toLowerCase();
}

m)
public static char getSub(String s, int n) {
    if(n > s.length() || n < 0){
        System.out.print("No good. Out of bounds");
        return;
    }
    return s.charAt(n);
}

n)
public static double inBetween(double d1, double d2) {
    return (d1+d2) / 2.0;
}

o)
public static boolean isAllEven(int n) {
    int temp = n;
    while(temp > 0) {
        if(temp % 2 == 1)
            return false;
        temp /= 10;
    }
    return true;
}

p)
public static boolean isAlpha(String s) {
    for(int i = 0; i < s.length(); i++)
        if(!Character.isAlphabetic(s.charAt(i)))
            return false;
    return true;
}

q)
```

```

public static boolean isAlphaNumeric(String s) {
    for(int i = 0; i < s.length(); i++)
        if(! (Character.isAlphabetic(s.charAt(i))
              || Character.isDigit(s.charAt(i)))) )
            return false;
    return true;
}

r)
public void dupe8(int n) {
    String num = n+"";
    for(int i = 0; i < num.length(); i++) {
        if(num.charAt(i) == '8')
            System.out.print("88");
        else
            System.out.print(num.charAt(i));
    }
}

s)
public int drop8(int n) {
    String num = n+"", ans = "";
    for(int i = 0; i < num.length(); i++) {
        if(num.charAt(i) != '8')
            ans += num.charAt(i);
    }
    return Integer.parseInt(ans);
}

t)
public void squareNum(int n) {
    String num = n+"";
    for(int i = 0; i < num.length(); i++) {
        int digit = Integer.parseInt(num.charAt(i)+"");
        System.out.print(digit*digit);
    }
}

u)
public static double frac(int n1, int n2) {
    if(n2 == 0) return -1;
    return (double)n1/n2;
}

v)
public static int inString(String s, char c) {
    int matches = 0;
    for(int i = 0; i < s.length(); i++)
        if(s.charAt(i) == c) matches++;
    return matches;
}

```

```
w)
public static int stringInString(String s, String t) {
    int matches = 0, t_len = t.length();
    for(int i = 0; i < s.length()-t_len; i++) {
        if( s.substring(i, i+t_len).equals(t) ) matches++;
    }
    return matches;
}
```

Problem 3:

a)

```
private static int series(int a, int b){
    int sum = 0;
    for(int i = a; i <= b; i++)
        sum += i;
    return sum;
}
```

b)

```
private static int series(int a, int b){
    int prod = 0;
    for(int i = a; i <= b; i++)
        prod *= i;
    return prod;
}
```

c)

```
private static int alternating(int a, int b){
    //bad since a needs to be positive
    if(a < 0 || b < a) return 0;

    int sum = 0;
    int counter = 0;
    for(int i = a; i <= b; i++){
        if(counter++ % 2 == 0) sum += i;
        else sum -= i;
    }
    return sum;
}
```

d)

```
private static double fractions(){
    double sum = 0.0;
    for(double i = 1.0; i <= 10.0; i += 1.0){
        sum += (1.0/(Math.pow(2.0,i)));
    }
    return sum;
}
```

e)

```

private static int odds(){
    int sum = 0;
    for(int i = 1; i <= 1000; i++) {
        sum += ((2*i)-1);
    }
    return sum;
}

```

Problem 4:

```

public class Problem4{
    private static int prodDigits(int n){
        int p = 1;
        if(n < 10) return n;

        while(n > 0){
            p *= n % 10;
            n = n / 10;
        }
        return p;
    }

    public static void main(String args[]){
        int n = 0;
        n = Integer.parseInt(args[0]);

        if(n <= 0)
            System.exit(1);

        System.out.println(prodDigits(n));
    } //main
} //class

```

Problem 5

```

public class Problem5{
    private static int sumOdd(int n){
        int p = 0;
        while(n > 0){
            if(n % 2 == 1) p += n%10;
            n = n/10;
        }
        return p;
    }

    public static void main(String args[]){
        int n = 0;
        n = Integer.parseInt(args[0]);

        if(n <= 0)
            System.exit(1);

        System.out.println(sumOdd(n));
    }
}

```

```
    } //main
} //class
```

Problem 6:

```
public class Problem6{
    private static void printChar(int n, String c){
        for(int i = 0; i < n; i++){
            for(int j = 0; j < n; j++){
                System.out.print(c);
            }
            System.out.println();
        }
    }
    public static void main(String args[]){
        int n = 0;
        String c = "";

        n = Integer.parseInt(args[0]);
        c = args[1];

        if(n <= 0) System.exit(1);

        System.out.println(printChar(n, c));
    } //main
} //class
```

Problem 7:

```
public class Problem7{
    private static int vowelCount(String s){
        int c = 0;
        for(int i = 0; i < s.length(); i++){
            if(s.charAt(i) == 'A' || s.charAt(i) == 'E'
               || s.charAt(i) == 'I' || s.charAt(i) == 'O'
               || s.charAt(i) == 'U'){
                c++;
            }
        }
        return c;
    }
    public static void main(String args[]){
        String word = args[0];
        System.out.println(vowelCount(word));
    } //main
} //class
```

Problem 8:

```
public class Problem8{
    private static String toBinary(int n){
        String tans = "", ans = "";
        while(n > 0){
```

```

        if(n%2 == 0) tans += "0";
        else tans += "1";
        n = n/2;
    }

    int len = tans.length();
    for(int i = 0; i < len; i++){
        ans += tans.charAt(len-i-1);
    }

    return ans;
}
public static void main(String args[]){
    int n = Integer.parseInt(args[0]);

    if(n <= 0) System.exit(1);

    System.out.println(toBinary(n));

} //main
} //class

```

Problem 9:

```

public class Problem9{
    private static boolean isPalindrome(int n){
        int a = n % 10; //last
        n = n /10;
        int b = n % 10;
        n = n / 10;
        int c = n % 10; //middle
        n = n / 10;
        int d = n % 10;
        n = n / 10;
        int e = n % 10; //first

        if(a == e && b == d){
            //since middle digit does not matter
            return true;
        }

        return false;
    } //isPalindrome()
    public static void main(String args[]){
        Scanner s = new Scanner(System.in);

        System.out.print("Please enter an integer: ");
        int num = s.nextInt();

        if(num < 10000 || num >= 100000){
            //not the right length
            System.exit(1);
        }
    }
}

```

```

        }

    if(isPalindrome(num)) {
        System.out.println("It is a palindrome");
    }else{
        System.out.println("It is not a palindrome");
    }
} //main
} //class

```

Problem 10:

```

import java.util.Scanner;
public class Problem10{
    public static void printNums(int a, int b, int c){
        //print the 3 numbers
        System.out.println(a + " " + b + " " + c);

        //find the max
        if(a >= b && a >= c)
            System.out.println("MAX IS: " + a);
        else if(b >= a && b >= c)
            System.out.println("MAX IS: " + b);
        else
            System.out.println("MAX IS: " + c);

        //find the min
        if(a <= b && a <= c)
            System.out.println("MIN IS: " + a);
        else if(b <= a && b <= c)
            System.out.println("MIN IS: " + b);
        else
            System.out.println("MIN IS: " + c);
    }
    public static void main(String args[]){
        Scanner s = new Scanner(System.in);

        int n1 = 0, n2 = 0, n3 = 0;
        System.out.println("Please enter 3 integers, separated by a
space: ");
        //get the 3 numbers
        n1 = s.nextInt();
        n2 = s.nextInt();
        n3 = s.nextInt();

        //call the method for the tasks
        printNums(n1, n2, n3);
    } //main
} //class

```

Problem 11:

```
import java.util.Scanner;
public class Problem11{
    private static int drop3(int n) {
        String num = n+"", ans = "";
        for(int i = 0; i < num.length(); i++)
            if(! (num.charAt(i) == '3'))
                ans += num.charAt(i);
        return Integer.parseInt(ans);
    }
    public static void main (String args[]) {
        Scanner s = new Scanner(System.in);
        System.out.print("Enter a 7 digit integer: ");
        int num = s.nextInt();

        while(num < 1000000 || num > 9999999) {
            System.out.print("Try again: ");
            num = s.nextInt();
        }
        System.out.println( drop3(num) );
    } //main
} //class
```

Problem 12:

```
import java.util.Scanner;
public class Problem12{
    private static int combineNums(int n1, int n2, int n3) {
        String num = n1""+n2""+n3"", ans="";
        for(int i = 0; i < 3; i++) {
            ans += num.charAt(i);
            ans += num.charAt(i+3);
            ans += num.charAt(i+6);
        }
        return Integer.parseInt(ans);
    }
    public static void main (String args[]) {
        Scanner s = new Scanner(System.in);
        System.out.print("Enter three 3 digit integers: ");
        int n1 = s.nextInt();
        int n2 = s.nextInt();
        int n3 = s.nextInt();

        if(n1 < 100 || n2 < 100 || n3 < 100
        || n1 > 999 || n2 > 999 || n3 > 999)
            System.exit(1);

        System.out.println( combineNums(n1, n2, n3) );
    } //main
} //class
```

Problem 13:

```
import java.util.Scanner;
public class Problem13{
    private static int upX(int num, int x) {
        String tnum = num+"", answer = "";
        for(int i = 0; i < tnum.length(); i++) {
            int digit = Integer.parseInt(tnum.charAt(i)+"");
            if(!(digit + x >=10)) answer += (x+digit)+"";
        }
        return Integer.parseInt(answer);
    }
    public static void main (String args[]) {
        Scanner s = new Scanner(System.in);
        System.out.print("Enter a 6 digit positive number: ");
        int num = s.nextInt();

        while(num < 100000 || num > 999999) {
            System.out.print("Try again: ");
            num = s.nextInt();
        }

        System.out.print("Enter a number between 1 and 5: ");
        int X = s.nextInt();

        if(X < 1 || X > 5)
            System.exit(1);

        System.out.println(upX(num, X));
    } //main
} //class
```

Problem 14:

```
import javax.swing.JOptionPane;
public class Problem14{
    private static boolean scary(int n) { return (n%13==0); }

    public static void main (String args[]) {
        if(args.length == 0)
            System.exit(1);

        int n = Integer.parseInt(args[0]);

        if(scary(n)) {
            JOptionPane.showMessageDialog(null, "AHHH!");
        } else {
            System.out.println("Whew!");
        }
    } //main
} //class
```

Problem 15:

```
public class Problem15{
    private static String dupeString(String w1, String w2) {
        int len = w2.length();
        String ans = "", w1copy = w1;
        w1copy = w1copy.replace(w2, "-");

        for(int i = 0; i < w1copy.length(); i++) {
            if(w1copy.charAt(i) == '-')
                ans += w2 + w2;
            else
                ans += w1copy.charAt(i);
        }
        return ans;
    }
    private static String dropString(String w1, String w2) {
        String ans = "", w1copy = w1;
        w1copy = w1copy.replace(w2, "-");

        for(int i = 0; i < w1copy.length(); i++) {
            if(!(w1copy.charAt(i) == '-'))
                ans += w1copy.charAt(i);
        }
        return ans;
    }
    public static void main (String args[]) {
        String w1 = args[0];
        String w2 = args[1];

        System.out.println(dupeString(w1, w2));
        System.out.println(dropString(w1, w2));
    } //main
} //class
```

Problem 16:

```
import java.util.Scanner;
public class Problem16{
    private static double FtoC(double f) { return(f-32.0)*5.0/9.0; }

    private static double CtoF(double c) { return(c*9.0/5.0)+32.0; }

    private static double FtoK(double f) { return(FtoC(f)+273.15); }

    private static double CtoK(double c) { return (c + 273.15); }

    private static double KtoF(double k) {
        return (k-273.15) * (9.0/5.0) + 32;
    }

    private static double KtoC(double k) { return k-273.15; }
```

```

public static void main (String args[]) {
    Scanner s = new Scanner(System.in);
    String scale = s.next();

    while(! (scale.equalsIgnoreCase("F")
              || scale.equalsIgnoreCase("C")
              || scale.equalsIgnoreCase("K")) ) {
        System.out.print("Try again: ");
        scale = s.next();
    }

    System.out.println("Enter the temperature: ");
    double temp = s.nextDouble();

    if(scale.equalsIgnoreCase("F")) {
        System.out.println(temp + " deg F to C: "
                           + FtoC(temp));
        System.out.println(temp + " deg F to K: "
                           + FtoK(temp));
    } else if(scale.equalsIgnoreCase("C")) {
        System.out.println(temp + " deg C to F: "
                           + CtoF(temp));
        System.out.println(temp + " deg C to K: "
                           + CtoK(temp));
    } else if(scale.equalsIgnoreCase("K")) {
        System.out.println(temp + " deg K to C: "
                           + KtoC(temp));
        System.out.println(temp + " deg K to F: "
                           + KtoF(temp));
    }
}
} //main
} //class

```

Output Tracing

Problem 17:

1. The output is:

55

165

41

Problem 18:

1. The output is:

6.0

3.0

6.0

6.0 -- -3.0

2. If the method were changed to type void, there will be an error, since you are trying to place the return value of the method f() into some local variables in the method g(). Since it is void, it will not return anything.

Problem 19:

1. The output is:

$$9 + 9 = 18$$

$$18 + 9 = 27$$

$$27 + 9 = 36$$

$$36 + 9 = 45$$

2. The output is:

$$9 + 9 = 18$$

$$18 + 9 = 27$$

$$27 + 9 = 36$$

$$36 + 9 = 45$$

$$45 + 9 = 54$$

$$54 + 9 = 63$$

$$63 + 9 = 72$$

$$72 + 9 = 81$$

Problem 20:

1. The output is:

$$18 + 9 = 27$$

$$36 + 9 = 45$$

2. The output is:

$$18 + 9 = 27$$

$$36 + 9 = 45$$

$$54 + 9 = 63$$

$$72 + 9 = 81$$

Problem 21:

1. The output is:

-7

Problem 22:

1. The method func() is void and does not return a value. That is the error as you are trying to get a value from a void method.

Problem 23:

1. The output for each value is:
 - a) Please enter a positive integer value: 10
100
1020
8000000
 - b) Please enter a positive integer value: 100
10000
1110
80000000
 - c) Please enter a positive integer value: 100000
1000000
101010
OVERFLOW (large negative number here)
 - d) Please enter a positive integer value: 3
300
1013
2400000

Problem 24:

1. The output is:
MATCH!
MATCH!
VERY HAPPY!

Problem 25:

1. The output is:
 - a. Given args[0] = 0 and args[1] = 3:
0
 - b. Given args[0] = 7 and args[1] = -2:
-11

- c. Given args[0] = -2 and args[1] = -2:
-1

Problem 26:

1. The output is:
 - a. Given args[0] = 4 and args[1] = 4:
44
 - b. Given args[0] = 5 and args[1] = 8:
58

Problem 27:

1. The output is:
 - a. Given args[0] = 3 and args[1] = 8:
38868
 - b. Given args[0] = 8 and args[1] = 3:
{no output displayed}
 - c. Given args[0] = 0 and args[1] = 3:
012

Solutions to Chapter 9 Exercises

Problem 1:

```
//assuming variable t to hold the current call number
public static void boxes(int x, char y){
    if(x == t) return;

    for(int i = 0; i < x; i++)
        System.out.print(y);
    System.out.println();

    t++;
    boxes(x, y);
}
```

Problem 2:

```
public static void triangle(int x) {
    if(x == 0) return;
    else{
        for(int i = 0; i < x; i++)
            if(x%2==1) System.out.print("1");
            else System.out.print("0");
    }
}
```

```

        else System.out.print("0");

        System.out.println();
    }
triangle(x-1);
}

```

Problem 3:

```

public static void triangle(int x){
    if(x > 1) triangle(x-1);

    for(int i = 0; i < x; i++)
        if(x%2==1) System.out.print("1");
        else System.out.print("0");

    System.out.println();
}

```

Problem 4:

```

public static void printBack(String s, int i){
    if(i < 0) return;

    if(i % 2 == 0)
        System.out.print(s.charAt(i));

    printBack(s, i-1);
}

```

Problem 5:

```

//assuming variable lg to hold the current largest digit
public static int largestDigit(int n){
    if(n <= 0) return lg;
    else{
        int t = n%10;
        if(t >= lg) lg = t;
        return largestDigit(n/10);
    }
}

```

Problem 6:

```

public static void reverse(int x) {
    if (x < 10) System.out.print(x);
    else {
        System.out.print(x % 10);
        reverse(x/10);
    }
}

```

Problem 7:

```
public static int sum(int x) {  
    if (x < 10) return x;  
    return x % 10 + sum(x / 10);  
}
```

Problem 8:

```
public static int numDigits(int x) {  
    if (x < 10) return 1;  
    return 1 + numDigits(x / 10);  
}
```

Converting Recursive Methods

Problem 9:

```
public static void boxes(int x, char y){  
    for(int i = 0; i < x; i++){  
        for(int j = 0; j < x; j++)  
            System.out.print(y);  
        System.out.println();  
    }  
}
```

Problem 10:

```
public static void triangle(int x){  
    for(int i = 1; i <= x; i++){  
        for(int j = 1; j <= x-i+1; j++)  
            if(i%2==1) System.out.print("1");  
            else System.out.print("0");  
        System.out.println();  
    }  
}
```

Problem 11:

```
public static void triangle(int x){  
    for(int i = 1; i <= x; i++){  
        for(int j = 1; j <= i; j++)  
            if(i%2==1) System.out.print("1");  
            else System.out.print("0");  
        System.out.println();  
    }  
}
```

Problem 12:

```
public static void printBack(String s, int i){  
    for(int j = s.length()-1; j >= 0; j--)  
        if(j % 2 == 1)
```

```
        System.out.print(s.charAt(j));
    }
```

Problem 13:

```
public static int largestDigit(int n){
    int lg = 0;
    while(n > 0){
        int t = n % 10;
        if(t >= lg) lg = t;
        n = n / 10;
    }
    return lg;
}
```

Problem 14:

```
public static void reverse(int n){
    while(n > 0){
        int t = n % 10;
        System.out.print(t);
        n = n / 10;
    }
}
```

Problem 15:

```
public static int sum(int n){
    if(n < 10) return n;
    int s = 0;

    while(n > 0){
        int t = n % 10;
        s += t;
        n = n / 10;
    }
    return s;
}
```

Problem 16:

```
public static int numDigits(int n){
    if(n < 10) return 1;
    int s = 0;

    while(n > 0){
        s++;
        n = n / 10;
    }
    return s;
}
```

Recursion Output Tracing

Problem 17:

- a. The output is:

21302

- b. The output is:

321302

- c. The output is:

321302

Problem 18:

- a. The output is:

512

- b. The output is:

734

- c. The output is:

734

Problem 20:

- a. The output is:

5121

- b. The output is:

7348

- c. The output is:

73429

Problem 21:

- a. The output is:

2

4

6

8

10

12

14

16

18

- b. The program will overflow the stack and be an infinite recursion.

Problem 22:

1. The output is:

15

19

13 12 11 10 10 11 12 13 1

17 16 15 14 13 12 11 10 10 11 12 13 14 15 16 17 1

21 20 19 18 17 16 15 14 13 12 11 10 10 11 12 13 14 15 16 17 18 19 20 21 1

Problem 23:

1. The output is:

0 1 2 0 0 1 2 7 1 1

Problem 24:

1. The output is:

2 9

4 8

6 7

8 6

Problem 25:

1. The output is:

6

5

4

3

2

1

0

1

2

3

4

5

6

Problem 26:

1. The output is:

7

77

777
7777

Problem 27:

1. The output is:

7777
777
77
7

Problem 28:

1. The output is:

1
4
9
16
25
36
49

Problem 29:

1. The output is:

15
14
13
12
11

Problem 30:

1. The output is:

0691215

Problem 31:

1. The output is:

+++++
++++
+++
++
+

Problem 32:

1. The output is:

10

9

8

7

6

5

5

7

9

11

13

15

15

Problem 33:

1. The output is:

1213121

Problem 34:

1. The output is:

1121123

Problem 35:

1. The output is:

0

4

6

Problem 36:

1. The output is:

1

10

5

6

Problem 37:

- a. Given the input of $x = 2$:

$z = 13$

- b. Given the input of $x = 3$:

$z = 10$

- c. Given the input of $x = 10$:

$z = 61$

Solutions to Chapter 10 Exercises

One Dimensional Array Exercises

Problem 1:

- a) short shorts[] = new short[20];
- b) int nums[] = new int[100];
- c) floats grades[] = new float[11];
- d) double numsDbl[] = new double[50];
- e) char characters[] = new char[100];
- f) String student_names[] = new String[22];

Problem 2:

- a)

```
for (int x = 0; x < shorts.length; x++) shorts[x] = 0;
```
- b)

```
for (int x = 0; x < nums.length; x++) nums[x] = 0;
```
- c)

```
for (int x = 0; x < grades.length; x++) grades[x] = 0.0;
```
- d)

```
for (int x = 0; x < numsDbl.length; x++) numsDbl[x] = 0.0;
```
- e)

```
for (int x = 0; x < characters.length; x++) characters[x] = 'A';
```
- f)

```
for (int x = 0; x < student_names.length; x++) student_names[x] = "";
```

Problem 3:

- a)

```
for(int i = 0; i < 15; i++)
    arr[i] = 'B';
```
- b)

```
for(int i = 0; i < 20; i++)
```

```

        System.out.print(iarr[i] + " ");
c)
for(int i = 0; i < 20; i++)
    System.out.print(far[i] + " ");

d)
for(int i = 0; i < 100; i++)
    larr[i]--;
e)
int total = 0;
for(int i = 0; i < 25; i++)
    if(car[i] == 'b' || car[i] == 'B')
        total++;

```

Problem 4:

```

a)
int i = 0;
while(i < 25) {
    arr[i] = 'X';
    i++;
}

b)
int i = 0;
while(i < 125) {
    System.out.print(iarr[i] + " ");
    i++;
}

c)
int i = 0;
while(i < 20) {
    System.out.print(fl[i] + " ");
    i++;
}

d)
int i = 0;
while(i < 100) {
    larr[i]--;
    i--;
}

e)
int total = 0, i = 0;
while(i < 25) {
    if(car[i] == 'b' || car[i] == 'B')
        total++;
    i++;
}

```

```
}
```

```
System.out.print(total);
```

Writing Programs

Problem 5:

```
public class Problem5{
    public static void main (String args[]) {
        int arr[] = {1,2,6,3,45,6,5,3,7,9,7,65,7,9,0};
        int len = arr.length;

        int min = arr[0];
        for (int c = 1; c < len; c++)
            if (arr[c] < min) min = arr[c];

        int max = arr[0];
        for (int c = 1; c < len; c++)
            if (arr[c] > max) max = arr[c];

        System.out.println("The min value is: " + min);
        System.out.println("The max value is: " + max);

        int sum = 0;
        double avg = 0.0;

        for(int i = 0; i < len; i++)
            sum += arr[i];

        System.out.println("The total sum is: " + sum);
        System.out.println("The average is: "
                           + ((double)(sum)/len));
    } //main
} //class
```

Problem 6:

```
public class Problem6{
    public static void main (String args[]) {
        short arr[] = {2,5,2,2,5,3,2,3,5,3,4,5,8,5,5,2};

        short first = arr[0], second = arr[1];
        int len = arr.length;

        //already 1 occurrence since it is there already
        int totFirst = 1, totSec = 1;

        for(int i = 2; i < len; i++){
            if(first == arr[i]) totFirst++;
            else if(second == arr[i]) totSec++;
        }
    }
}
```

```

        if(totFirst > totSec){
            //double the elements
            for(int i = 0; i < len; i++)
                arr[i] *= 2;
        }else if(totSec > totFirst){
            //decrease the elements by 5
            for(int i = 0; i < len; i++)
                arr[i] -= 5;
        }

        for(int i = 0; i < len; i++)
            System.out.print( arr[i] + " ");
    } //main
} //class

```

Problem 7:

```

public class Problem7{
    public static void main (String args[]) {
        short nums[] = {18,912,121,300,601,2121,3122,1991,12,1001};

        int tot = 0;
        for(int i = 0; i < nums.length; i++) {
            int number = nums[i];
            while (number > 0) {
                if(number % 10 == 1)
                    tot++;
                number /= 10;
            }
        }
        System.out.println("There are " + tot
            + " ones in the array");
    } //main
} //class

```

Problem 8:

```

import java.util.Scanner;
public class Problem8{
    public static void main(String args[]){
        Scanner s = new Scanner(System.in);
        String animals[] = {"Dog", "Cat", "Lion", "Tiger",
                            "Lizard", "Bear", "Penguin", "Gorilla"};

        System.out.print("Enter an integer > 0: ");
        int x = s.nextInt();

        if(x < 0)
            System.exit(1);

        for(int i = 0; i < animals.length; i++) {
            if(animals[i].length() <= x)

```

```

        System.out.print(" ");
    else
        System.out.print(animals[i].charAt(x));
    }
} //main
} //class

```

Problem 9:

```

import java.util.Scanner;
public class Problem9{
    public static void main(String args[]){
        Scanner s = new Scanner(System.in);
        int arr[] =
{100,221,68,90,95,96,85,73,77,79,77,65,97,99,101};

        System.out.print("Enter an integer between 10 & 25: ");
        int x = s.nextInt();

        while(x < 10 || x > 25) {
            System.out.print("Try again: ");
            x = s.nextInt();
        }

        int first = arr[0];
        int top = first + x;
        int bot = first - x;

        for(int i = 1; i < arr.length; i++) {
            if(arr[i] >= bot && arr[i] <= top)
                System.out.print(arr[i] + " ");
        }
    } //main
} //class

```

Output Tracing

Problem 10:

a)
910

b)
81 64 49 36

c)
C D E A E

d)
AB BC CD DA AB BC CD DA

e)
HoImG

f)
HoHoHoHoHo

Problem 11:

The output is:

77
4N
0N
2N
3N
1N
6N

Problem 12:

The output is:

1
6
0

Problem 13:

The output is:

4.0 2.1 3.3
3.3
4.0
3.3
3.3
4.4
5.5

Problem 14:

The output is:

Before: selppA
After: seLpPA

Two Dimensional Array Exercises

Problem 1:

- a) int matrix[][] = new int[5][5];
- b) float scores[][] = new float[10][4];
- c) char picture[][] = new char[12][12];
- d) double something[][] = new double[4][6];
- e) String str[][] = new String[10][100];

Problem 2:

- a)

```
for(int i = 0; i < 10; i++)
    for(int j = 0; j < 5; j++)
        li[i][j]++;
```
- b)

```
double min = darr[0];
for(int i = 0; i < 3; i++) {
    for(int j = 0; j < 5; j++) {
        if(darr[i][j] < min)
            min = darr[i][j];
    }
}
```
- c)

```
int vowels = 0;
for(int r = 0; r < 10; r++) {
    for(int c = 0; c < 4; c++) {
        if(letters[r][c] == 'a' || letters[r][c] == 'e'
           || letters[r][c] == 'i' || letters[r][c] == 'o'
           || letters[r][c] == 'u')
            vowels++;
    }
}
```
- d)

```
int negatives = 0;
for(int r = 0; r < 6; r++) {
    for(int c = 0; c < 6; c++) {
        if(vals[r][c] < 0.0) negatives++;
    }
}
```
- e)

```
int product = 1;
for(int r = 0; r < 4; r++) {
    for(int c = 0; c < 2; c++) {
        if(chart[r][c] > 0)
            product *= chart[r][c];
    }
}
```

Problem 3:

a)

```
int i = 0, j = 0;
while(i < 10){
    while(j < 5){
        dd[i][j] += 1.0;
        j++;
    }
    j = 0;
    i++;
}
```

b)

```
int min = arr[0], i = 0, j = 0;
while(i < 3){
    while(j < 5){
        if(arr[i][j] < min)
            min = arr[i][j];
        j++;
    }
    j = 0;
    i++;
}
```

c)

```
int r = 0, c = 0, i = 0, lower=0;
while(r < 2) {
    c = 0;
    while(c < 2) {
        i=0;
        while(i < words[r][c].length()) {
            if(Character.isLowerCase(words[r][c].charAt(i)))
                lower++;
            i++;
        }
        c++;
    }
    r++;
}
```

d)

```
int r = 0, c = 0, high = 0, low = 0;
while(r < 5) {
    c = 0;
    while(c < 5) {
        if(temperatures[r][c] > high)
            high = temperatures[r][c];
        if(temperatures[r][c] < low)
            low = temperatures[r][c];
        c++;
    }
}
```

```

        r++;
    }

e)
int r = 0, c = 0;
while(r < 3) {
    c = 0;
    while(c < 20) {
        numbers[r][c] += 2.667;
        c++;
    }
    r++;
}

```

Problem 4:

```

public class Problem4{
    private static long numOnes(long x) {
        if (x <= 0) return 0;
        else if(x % 10 == 1) return 1 + numOnes(x / 10);
        else return 0 + numOnes(x / 10);
    }
    public static void main(String args[]){
        long nums[][] = {{18212, 91112, 2056, 3002, 60121},
                        {21121, 319922, 199921, 120092, 1014401},
                        {98874, 987321, 21245, 652, 2211}};

        for(int i = 0; i < 3; i++)
            for(int j = 0; j < 5; j++)
                System.out.print( nums[i][j] + " has "
                                + numOnes(nums[i][j]) );
    } //main
} //class

```

Problem 5:

```

public class Problem5{
    private static int findMin(int arr[][], int row){
        int min = arr[row][0];
        for(int i = 1; i < 4; i++)
            if(arr[row][i] < min) min = arr[row][i];
        return min;
    }
    private static int getSumEven(int arr[][]){
        int sum = 0;
        for(int i = 0; i < 4; i++)
            for(int j = 0; j < 4; j++)
                if(arr[i][j] % 2 == 0)
                    sum += arr[i][j];
        return sum;
    }
    private static boolean isPrime(int n){

```

```

        if(n % 2 == 0) return false;
        if(n == 2) return true;

        for(int i = 3; i < n; i++)
            if(n % i == 0) return false;

        return true;
    }
    private static int getPrimeCount(int arr[][]){
        int total = 0;
        for(int i = 0; i < 4; i++)
            for(int j = 0; j < 4; j++)
                if(isPrime(arr[i][j]))
                    total++;
        return total;
    }
    public static void main (String args[]) {
        int arr[][] = {{1,3,6,8},{7,5,3,4},{5,76,8,9},{78,6,4,11}};

        //find the minimum in each row
        for(int i = 0; i < 4; i++)
            System.out.print("Minimum for row " + i + " is: "
                + findMin(arr, i) );

        System.out.print("The sum of the even numbers is: "
            + getSumEven(arr) );

        System.out.print("The total number of prime numbers are: "
            + getPrimeCount(arr) );
    } //main
} //class

```

Problem 6:

```

public class Problem6{
    private static int findMax(int arr[][], int row){
        int max = arr[row][0];

        for(int i = 1; i < 4; i++)
            if(arr[row][i] > max) max = arr[row][i];

        return max;
    }
    private static int getProdOdd(int arr[][]){
        int prod = 1;
        for(int i = 0; i < 4; i++)
            for(int j = 0; j < 4; j++)
                if(arr[i][j] % 2 == 1)
                    prod *= arr[i][j];

        return prod;
    }
}

```

```

private static int findMinCol(int arr[][], int col){
    int min = arr[0][col];

    for(int i = 1; i < 4; i++)
        if(arr[i][col] < min) min = arr[i][col];

    return min;
}
public static void main (String args[]) {
    int arr[][] = {{1,3,6,8},{7,5,3,4},{5,76,8,9},{78,6,4,11}};

    //find the maximum in each row
    for(int i = 0; i < 4; i++)
        System.out.print("Maximum for row " + i + " is: "
            + findMax(arr, i) );

    System.out.print("The product of the odd numbers is: "
        + getProdOdd(arr) );

    //find the minimum in each column
    for(int i = 0; i < 4; i++)
        System.out.print("Minimum for column " + i + " is: "
            + findMinCol(arr, i) << endl;

} //main
} //class

```

Problem 7:

```

import java.util.Scanner;
public class Problem7{
    public static void main(String args[]){
        Scanner s = new Scanner(System.in);
        int array[][] = new int[3][3];

        System.out.println("Enter 9 integers: ");
        for(int i = 0; i < 3; i++) {
            for(int j = 0; j < 3; j++) {
                array[i][j] = s.nextInt();
            }
        }

        for(int i = 0; i < 3; i++) {
            for(int j = 0; j < 3; j++) {
                for(int c = 0; c < array[i][j]; c++) {
                    System.out.print("X");
                }
                System.out.print(" ");
            }
            System.out.println();
        }
    } //main
} //class

```

```
} //class
```

Problem 8:

```
import java.util.Scanner;
public class Problem8{
    public static void main(String args[]){
        Scanner s = new Scanner(System.in);
        double entries[][] = new double[2][5];

        System.out.println("Enter 10 double precision values: ");
        for(int i = 0; i < 2; i++) {
            for(int j = 0; j < 5; j++) {
                entries[i][j] = s.nextDouble();
            }
        }

        for(int i = 0; i < 1; i++) {
            for(int j = 0; j < 5; j++) {
                if(entries[i][j] >= entries[i+1][j])
                    System.out.print(entries[i][j] + " ");
                else
                    System.out.print(entries[i+1][j] + " ");
            }
        }
    } //main
} //class
```

Problem 9:

```
public class Problem9{
    public static void main(String args[]){
        Scanner s = new Scanner(System.in);
        String array[][] = {{"a", "orange", "apple", "banana"}, 
                            {"s", "Mississippi", "Missouri", "Florida"}, 
                            {"3","123345323","331231","90876451"}};

        //print before
        for(int i = 0; i < 3; i++) {
            for(int j = 1; j < 4; j++) {
                System.out.print(array[i][j] + " ");
            }
            System.out.println();
        }

        //remove elements
        for(int i = 0; i < 3; i++) {
            char lookFor = array[i][0].charAt(0);

            for(int j = 1; j < 4; j++) {
                String word = array[i][j];
                if(word.charAt(0) == lookFor)
                    System.out.print(word + " ");
            }
            System.out.println();
        }
    }
}
```

```

        String ans = "";

        for(int c = 0; c < word.length(); c++) {
            if(! (word.charAt(c) == lookFor))
                ans += word.charAt(c);
        }

        array[i][j] = ans;
    }
}

System.out.println();
//print after
for(int i = 0; i < 3; i++) {
    for(int j = 1; j < 4; j++) {
        System.out.print(array[i][j] + " ");
    }
    System.out.println();
}
} //main
} //class

```

Arrays and Methods

Problem 1:

- a) public static double findMax(double arr[]);
- b) public static boolean isThere(int arr[], int element);
- c) public static void print(int arr[]);
- d) public static void swapElements(int arr[], int swap1, int swap2);
- e) public static void reverseElements(double arr[]);
- f) public static void reverseChar(char c[]);
- g) public static float addArraySum(float arr1[][][], float arr2[][][]);
- h) public static int findSum(int arr[]);
- i) public static double findPosPercent(short arr[]);
- j) public static void removeCapitals(String arr[][]);
- k) public static int passingScore(int arr[], int num);
- l) public static void decreaseByX(int arr[], int x);
- m) public static void printCaps(String arr[][]);
- n) public static void printEvens(short arr[]);
- o) public static void printLarger(double arr[]);
- p) public static void printASCII(char arr[]);
- q) public static void replaceDigitWith(String arr[][]);
- r) public static void invertNumbers(short arr[][]);
- s) public static int findPositive2D(double arr[][]);
- t) public static int findProdNegative(int arr[][]);

Problem 2:

- a)
- ```

public static double findMax(double arr[]) {
 int max = 0;

```

```

 for(int i = 0; i < arr.length; i++){
 if(arr[i] > max) max = arr[i];
 }
 return max;
 }

b)
public static boolean isThere(int arr[], int element){
 for(int i = 0; i < arr.length; i++){
 if(arr[i] == element) return true;
 }
 return false;
}

c)
public static void print(int arr[]){
 for(int i = 0; i < arr.length; i++){
 System.out.print(arr[i] + " ");
 }
}

d)
public static void swapElements(int arr[], int swap1, int swap2){
 int temp = arr[swap1];
 arr[swap1] = arr[swap2];
 arr[swap2] = temp;
}

e)
public static void reverseElements(double arr[]){
 int x = arr.length-1;
 for(int i = 0; i < n; i++, x--){
 if(i >= x) break;

 double t = arr[i];
 arr[i] = arr[x];
 arr[x] = t;
 }
}

f)
public static void reverseChar(char c[]){
 int x = c.length-1;
 for(int i = 0; i < n; i++, x--){
 if(i >= x) break;

 char t = c[i];
 c[i] = c[x];
 c[x] = t;
 }
}

```

```

g)
public static float addArraySum(float arr1[][][], float arr2[][][]) {
 float total = 0.0;
 for(int i = 0; i < 5; i++) {
 for(int j = 0; j < 5; j++) {
 total += arr1[i][j];
 total += arr2[i][j];
 }
 }
 return total;
}

h)
public static int findSum(int arr[]) {
 int sum = 0;
 for(int i = 0; i < arr.length; i++)
 if(arr[i] >= 0) sum+= arr[i];
 return sum;
}

i)
public static double findPosPercent(short arr[]) {
 double tot = 0.0;
 int len = arr.length;
 for(int i = 0; i < len; i++)
 if(arr[i] >= 0) tot+=1.0;
 return (double) (tot/len);
}

j)
public static void removeCapitals(String arr[][][]) {
 for(int i = 0; i < 12; i++) {
 for(int j = 0; j < 6; j++) {
 String ans = "", word = arr[i][j];
 for(int c = 0; c < word.length(); c++) {
 if(!Character.isUpperCase(word.charAt(c)))
 ans += word.charAt(c);
 }
 arr[i][j] = ans;
 }
 }
}

k)
public static int passingScore(int arr[], int num) {
 int tot = 0;
 for(int i = 0; i < arr.length; i++)
 if(arr[i] >= num) tot++;
 return tot;
}

```

```

l)
public static void decreaseByX(int arr[], int x) {
 for(int i = 0; i < arr.length; i++)
 arr[i] -= x;
}

m)
public static void printCaps(String arr[][][]) {
 for(int i = 0; i < 4; i++) {
 for(int j = 0; j < 4; j++) {
 String word = arr[i][j];
 for(int c = 0; c < word.length(); c++)
 if(Character.isUpperCase(word.charAt(c)))
 System.out.print(word.charAt(c));
 }
 }
}

n)
public static void printEvens(short arr[]) {
 for(int i = 0; i < arr.length; i++)
 if(arr[i] % 2 == 0)
 System.out.print(arr[i]);
}

o)
public static void printLarger(double arr[]) {
 for(int i = 0; i < arr.length; i+=2)
 if(arr[i] >= arr[i+1])
 System.out.print(arr[i]);
 else
 System.out.print(arr[i+1]);
}

p)
public static void printASCII(char arr[]) {
 for(int i = 0; i < arr.length; i++)
 System.out.print((int)arr[i] + " ");
}

q)
public static void replaceDigitWith(String arr[][][]) {
 for(int i = 0; i < 20; i++) {
 for(int j = 0; j < 20; j++) {
 String ans = "", word = arr[i][j];
 for(int c = 0; c < word.length(); c++) {
 if(Character.isDigit(word.charAt(c)))
 ans += '*';
 else
 ans += word.charAt(c);
 }
 }
 }
}

```

```

 arr[i][j] = ans;
 }
}

r)
public static void invertNumbers(short arr[][]){
 for(int i = 0; i < 10; i++){
 for(int j = 0; j < 3; j++)
 arr[i][j] *= -1;
 }
}

s)
public static int findPositive2D(double arr[][]){
 int tot = 0;
 for(int i = 0; i < 5; i++) {
 for(int j = 0; j < 100; j++)
 if(arr[i][j] >= 0.0) tot++;
 }

 return tot;
}

t)
public static int findProdNegative(int arr[][]){
 int tot = 1;
 for(int i = 0; i < 50; i++) {
 for(int j = 0; j < 10; j++)
 if(arr[i][j] < 0) tot *= arr[i][j];
 }

 return tot;
}

```

### Writing Programs

#### **Problem 3:**

```

import java.util.Scanner;
public class Problem3{
 //find the max number
 public static int findMax(int arr[]){
 int m = arr[0];
 for(int i = 1; i < arr.length; i++)
 if(arr[i] > m) m = arr[i];

 return m;
 }
 //find the min number
 public static int findMin(int arr[]){
 int m = arr[0];
 for(int i = 1; i < arr.length; i++)

```

```

 if(arr[i] < m) m = arr[i];

 return m;
}
//find the sum
public static int getTotal(int arr[]){
 int sum = 0;
 for(int i = 0; i < arr.length; i++)
 sum += arr[i];

 return sum;
}
//find the average
public static double getAvg(int arr[]){
 double sum = (double) getTotal(arr);
 double len = (double) arr.length;

 return (sum/len);
}
public static void main(String args[]){
 Scanner s = new Scanner(System.in);
 int n = 0;

 System.out.println("Please enter an integer n: ");
 n = s.nextInt();

 if(n < 1) //not enough of a size
 System.exit(1);

 //declare the array of size n
 int arr[] = new int[n];

 //get the numbers
 for(int i = 0; i < arr.length; i++){
 System.out.println("Please enter number " + i + ": ");
 arr[i] = s.nextInt();
 }
 System.out.println("The max number is: " + findMax(arr));
 System.out.println("The min number is: " + findMin(arr));
 System.out.println("The total sum is: " + getTotal(arr));
 System.out.println("The average is: " + getAvg(arr));
} //main
} //class

```

#### **Problem 4:**

```

import java.util.Scanner;
public class Problem4{
 //size of the array
 private static int RSIZE = 4, CSIZE = 4;

 //array itself

```

```

private static int arr[][];

private static int findMin(int row) {
 int min = arr[row][0];

 for(int i = 1; i < CSIZE; i++)
 if(arr[row][i] < min) min = arr[row][i];

 return min;
}

private static String displayEven() {
 String s = "";
 for(int i = 0; i < RSIZE; i++) {
 for(int j = 0; j < CSIZE; j++)
 if(arr[i][j] % 2 == 0)
 s += arr[i][j] + " ";
 }
 s += "\n";
 return s;
}

private static int getArraySum() {
 int sum = 0;
 for(int i = 0; i < RSIZE; i++)
 for(int j = 0; j < CSIZE; j++)
 sum += arr[i][j];

 return sum;
}

private static String print() {
 String s = "";
 for(int i = 0; i < RSIZE; i++) {
 for(int j = 0; j < CSIZE; j++)
 s += arr[i][j] + " ";
 }
 s += "\n";
 return s;
}

public static void main(String args[]) {
 Scanner s = new Scanner(System.in);

 //initialize the array
 arr = new int[4][4];

 //enter the numbers
 for(int i = 0; i < 4; i++)
 for(int j = 0; j < 4; j++)
 System.out.print("Enter number for " + i + ", "
 + j + ": ");

 //do the tasks
 String str = "";
}

```

```

//first, find the min in each row of the array
for(int i = 0; i < RSIZE; i++) {
 str += "Row " + i + " min value: " + findMin(i) + "\n";
}

//display the min message
System.out.println(str);

//find the even numbers
System.out.println("Even numbers in array are: "
+ displayEven());

//find the sum in the array
System.out.println("Sum of numbers is: " + getArraySum());
} //main
} //class

```

### **Problem 5:**

```

import java.util.Scanner;
public class Problem5{
 private static void printGrid(short arr[]) {
 for(int i = 0; i < arr.length; i++) {
 short val = arr[i];
 for(int j = 1; j <= 5; j++) {
 if(j == val) System.out.print("X");
 else System.out.print("-");
 }
 System.out.println();
 }
 }
 public static void main (String args[]) {
 Scanner s = new Scanner(System.in);
 short entries[] = new short[5];

 System.out.println("Enter 5 numbers: ");

 for(int i = 0; i < 5; i++)
 entries[i] = s.nextShort();

 printGrid(entries);
 } //main
} //class

```

### **Problem 6:**

```

import java.util.Scanner;
public class Problem6{
 private static int returnNum(int[] arr) {
 int pos = 10000000, ans = 0;
 for(int i = 0; i < arr.length; i++) {

```

```

 ans += arr[i] * pos;
 pos /= 10;
 }
 return ans;
}
private static int returnRevNum(int[] arr) {
 int pos = 10000000, ans = 0;
 for(int i = 7; i >= 0; i--) {
 ans += arr[i] * pos;
 pos /= 10;
 }
 return ans;
}
public static void main (String args[]) {
 Scanner s = new Scanner(System.in);
 int entries[] = new int[8];

 System.out.println("Enter 8 numbers between 0 and 9: ");

 for(int i = 0; i < 8; i++) {
 entries[i] = s.nextInt();
 if(entries[i] < 0 || entries[i] > 9)
 System.exit(1);
 }

 System.out.println(returnNum(entries));
 System.out.println(returnRevNum(entries));
} //main
} //class

```

### Problem 7:

```

import java.util.Scanner;
public class Problem7{
 private static int smallestProd(int[] arr) {
 int min = arr[0]*arr[1];
 for(int i = 2; i < 10; i+=2) {
 if(arr[i]*arr[i+1] <= min) min = arr[i]*arr[i+1];
 }
 return min;
 }
 private static int biggestDiff(int[] arr) {
 int max = 0;
 for(int i = 0; i < 10; i+=2) {
 if(arr[i]-arr[i+1] >= max) max = arr[i]-arr[i+1];
 }
 return max;
 }
 private static int biggestSum(int[] arr) {
 int max = 0;
 for(int i = 0; i < 10; i+=2) {
 if(arr[i]+arr[i+1] >= max) max = arr[i]+arr[i+1];
 }
 }
}

```

```

 }
 return max;
 }
 public static void main (String args[]) {
 Scanner s = new Scanner(System.in);
 int entries[] = new int[10];

 System.out.println("Enter 10 numbers: ");

 for(int i = 0; i < 10; i++)
 entries[i] = s.nextInt();

 System.out.println(smallestProd(entries));
 System.out.println(biggestDiff(entries));
 System.out.println(biggestSum(entries));
 } //main
} //class

```

### Output Tracing

#### **Problem 8:**

a)

1A

2C

3E

4G

I

73 2 3 4

b)

1A

2C

3E

G

71 2 3 4

c)

m h e o l

mheolloehm

d)

\*\*\*\*

\*\*\*

\*\*\*\*\*  
\*\*  
\*\*\*

e)  
\*\*\*  
\*\*  
\*\*\*\*\*  
\*\*\*  
\*\*\*

f)  
4  
5  
2  
8  
8  
2  
5  
4

**Problem 9:**

The output is:

10 9 8 7 6 5 4 3 2 1  
1 2 3 4 5 5 4 3 2 1

**Problem 10:**

The output is:

10 9 8 7 6 5 4 3 2 1  
1 2 3 4 5 6 7 8 9 10

**Problem 11:**

The output is:

Allan A  
Peter AP  
Brian APB

Sally APBS  
Susan APBSS

**Problem 12:**

The output is:

3 4 5 6 7 8 9  
4 5 6 7 8 9  
5 6 7 8 9  
6 7 8 9  
7 8 9  
8 9  
9  
H H H H H  
H H H H H  
e e e e  
e e e e  
1 1 1  
1 1 1  
1 1  
1 1  
o  
o

**Problem 13:**

1. XXXXXXXXXXXXXXX
2. YYYYYYYYYYYYYYYYYYYYYYYYY
3. ZZZZZZG

**Problem 14:**

The output is:

A B C D E F G H I J  
B C D E F G H I J K K L M N O  
C D E F G H I J K L L M N O P

**Problem 15:**

1. Value of counter1 is: 1
2. Value of counter2 is: 3
3. The output is:

4 5 6 7

4 5

Sum: 16

### Solutions to Chapter 11 Exercises

#### **Problem 1:**

```
import java.util.Random;
public class Problem1{
 public static void main(String args[]){
 Random r = new Random();
 System.out.println("The number is: "
 + r.nextInt(10) + r.nextInt(10) + r.nextInt(10));
 }
}
```

#### **Problem 2:**

```
import java.util.Random;
public class Problem2{
 public static void main(String args[]){
 Random r = new Random();
 System.out.println("The number is: "
 + r.nextInt(10) + r.nextInt(10)
 + r.nextInt(10) + r.nextInt(10));
 }
}
```

#### **Problem 3:**

```
import java.util.Random;
public class Problem3{
 public static void main(String args[]){
 Random r = new Random();
 int count = 0, num = 0;
 while(count < 10){
 num = r.nextInt(50) + 1;
 while(num % 2 == 1)
```

```

 num = r.nextInt(50) + 1;

 System.out.println(num);
 count++;
 }

} //main
} //class

```

**Problem 4:**

```

import java.util.Random;
public class Problem4{
 public static void main(String args[]){

 Random r = new Random();
 int count = 0, num = 0;
 while(count < 10){
 num = r.nextInt(100) + 1;
 while(num % 2 == 0)
 num = r.nextInt(100) + 1;
 System.out.println(num);
 count++;
 }

 } //main
} //class

```

**Problem 5:**

```

import java.util.Random;
public class Problem5{
 public static void main(String args[]){
 Random r = new Random();

 int count = 0, num = 0;
 for(int i = 0; i < 50; i++){
 num = r.nextInt(1001);
 if(num % 2 == 0 && num % 5 == 0){
 count++;
 System.out.println(num);
 }
 }

 System.out.println("Total of: " + count
 + " divisible by 2 and 5.");
 } //main
} //class

```

**Problem 6:**

```

import java.util.Random;
public class Problem6{

```

```

public static void main(String args[]){
 Random r = new Random();

 int count = 0, num = 0;
 for(int i = 0; i < 100; i++) {
 num = r.nextInt(777) + 1;
 if(num % 10 == 7) {
 count++;
 System.out.println(num);
 }
 }

 System.out.println("Total of: " + count
 + " ending in 7.");
} //main
} //class

```

**Problem 7:**

```

import java.util.Random;
public class Problem7{
 public static void main(String args[]){
 Random r = new Random();

 short arr[] = new short[20];

 //generate numbers
 for(int i = 0; i < 20; i++)
 arr[i] = r.nextInt(50) + 1;

 //find mean
 double avg = 0.0;
 for(int i = 0; i < 20; i++)
 avg += (double)arr[i];

 avg /= 20.0;

 System.out.println("The average is: " + avg);
 } //main
} //class

```

**Solutions to Chapter 13 Exercises**

*A main method and a sample class are provided for testing only:*

**Problem 1:**

```

public class Square{

 private int side;

```

```

//constructors:
public Square(){
 side = 1;
}

public Square(int s){
 side = s;
}

public int setSide(int s){
 if(s < 1){
 return;
 }
 side = s;
}

public int getSide(){ return side; }

//member methods:
public int perimeter(){ return 4*side; }

public int area(){ return side*side; }
}

public class Problem1{
 public static void main(String args[]){
 Square s = new Square(5);
 System.out.println("Side: " + s.getSide());

 s.setSide(10);

 System.out.println("Side: " + s.getSide());

 System.out.println("Area: " + s.perimeter());
 System.out.println("Perimeter: " + s.area());
 }
}

```

### **Problem 2:**

```

public class Book{
 private String title, author;
 private double price;
 private int year;

 public Book(){
 title = "";
 author = "";
 price = 0.0;
 year = 0;
 }
 public Book(String t, String a, double p, int y){

```

```

 title = t;
 author = a;
 price = p;
 year = y;
 }

 public String getTitle(){ return title; }
 public String getAuthor(){ return author; }
 public double getPrice(){ return price; }
 public int getYear(){ return year; }

 public void setTitle(String t){ title = t; }
 public void setAuthor(String a){ author = a; }
 public void setPrice(double p){ price = p; }
 public void setYear(int y){ year = y; }

 public int compare(Book b){
 if(price < b.price) return -1;
 else if(price == b.price) return 0;

 return 1;
 }
 public String toString(){
 String ans = "";
 ans += "Title: " + title + "\n";
 ans += "Author: " + author + "\n";
 ans += "Price: $" + price + "\n";
 ans += "Published: " + year + "\n";

 return ans;
 }
}

public class Problem2{
 public static void main(String args[]){
 Book b1 = new Book("Peace", "Somebody", 33.30, 1900);
 Book b2 = new Book("The Beach", "Joe Doe", 25.00, 1994);

 b1.setPrice(34.95);
 b2.setYear(1999);

 //should be 1:
 System.out.println(b1.compare(b2) + "\n");

 System.out.println(b1);
 System.out.println(b2);
 }
}

```

## Solutions to Chapter 14 Exercises

### **Problem 1:**

1. The super class is Math.
2. The inherited methods are square(), cube(), pow() and factorial().
3. The output is:

120

25

125

125

295

### **Problem 2:**

1. The super class is Team

2. The inherited methods are getName(), getMembers(), setName(), memberJoined() and memberLeft().

3. The output is:

Mets player joined!

Mets player left!

Mets player joined!

Mets player joined!

Mets player joined!

Mets player left!

Mets player joined!

Total addition are: 3

### **Problem 3:**

1. Fruit is the superclass.

2. The methods pick(), sell() and getSales() are all inherited from the super class.

3. The value is 0. It generated the message "Can't sell it!"

4. The output is:

Apples sold: \$4.36

Can't sell it!  
Oranges sold: \$1.78

**Problem 4:**

1. The two subclasses are Dog and Cat.
2. The variable count is simply being used to keep track of the number of objects.
3. There are 3 instances of the class Dog.

4. The output is:

lice is 4 years old!  
teve is 6 years old!  
ERRY is 100 years old!  
ax is 5 years old!  
alph is 2 years old!

**Problem 5:**

1. The subclass is class B.

2. The inherited methods are multiply() and squareX().

3. The output is:

14  
55  
A: x= 3 y= 30  
B: xx= 10 yy= 5  
A: x= 3 y= 30  
B: xx= 9 yy= 4  
A: x= 3 y= 30  
B: xx= 8 yy= 3  
B: xx= 7 yy= 2  
B: xx= 6 yy= 1

*Solutions to Chapter 16 Exercises*

**Problem 1:**

- a) This is the library for file input. You will still need to import java.io.FileWriter in order to get it to work properly as that is the other piece of the puzzle.

- b) This is the library to handle all aspects of file input and output.
- c) This method will close either the BufferedReader or the BufferedWriter.
- d) This method will read a line of a file when dealing with file input
- e) (BufferedReader).
- f) This method will write to a file (BufferedWriter).

**Problem 2:**

```

import java.io.*;
import java.util.*;

public class Problem2{
 private static boolean isPrime(int n){
 if(n % 2 == 0) return false;
 if(n == 2) return true;

 for(int i = 3; i < n; i++)
 if(n % i == 0) return false;

 return true;
 }

 public static void main (String args[]) {
 String line = "";
 int count = 0;

 try{
 BufferedReader br = new BufferedReader(
 new FileReader("input.txt"));

 line = br.readLine();
 StringTokenizer st = new StringTokenizer(line);

 for(int i = 0; i < 10; i++){
 int num = Integer.parseInt(st.nextToken());
 if(isPrime(num))
 count++;
 }

 System.out.println("There are " + count
 + " primes in the file!");
 }catch(Exception e){}
 } //main
} //class

```

**Problem 3:**

```

import java.io.*;
import java.util.*;

```

```

public class Problem3{
 private static int fib(int n){
 if(n <= 2) return 1;
 return fib(n-2) + fib(n-1);
 }

 public static void main (String args[]) {
 String line = "";
 int count = 0;
 int fibs[] = new int[10];

 for(int j = 1; j <= 10; j++)
 fibs[j-1] = fib(j);

 try{
 BufferedReader br = new BufferedReader(
 new FileReader("input.txt"));

 line = br.readLine();
 StringTokenizer st = new StringTokenizer(line);

 for(int i = 0; i < 10; i++){
 int num = Integer.parseInt(st.nextToken());

 //find the number of fibs in the file:
 for(int j = 0; j < 10; j++)
 if(num == fibs[j]){ //found a match
 count++;
 }
 }

 System.out.println("There are " + count
 + " fibs in the file!");
 }catch(Exception e){}
 } //main
} //class

```

#### **Problem 4:**

```

import java.io.*;
public class Problem4{
 private static void sort(String[] words){
 //selection sort the array:
 for(int i = 0; i < words.length-1; i++){
 int low = i;
 for(int j = i+1; j < words.length; j++){
 if(words[j].compareToIgnoreCase(words[low]) < 0)
 low = j;
 }
 String t = words[low];
 words[low] = words[i];
 words[i] = t;
 }
 }
}

```

```
 words[i] = t;
 }
}
public static void main (String args[]) {
 String line = "";
 int count = 0;
 String words[] = new String[10];

 try{
 BufferedReader br = new BufferedReader(
 new FileReader("input.txt"));

 for(int i = 0; i < 10; i++){
 words[i] = br.readLine();
 }

 br.close();
 sort(words);

 BufferedWriter bw = new BufferedWriter(
 new FileWriter("output.txt"));

 for(int i = 0; i < 10; i++){
 bw.write(words[i] + "\n");
 }

 bw.close();
 } catch(Exception e) {}

 } //main
} //class
```