

Java Language Summary

28th June 2004

Ethan Metsger <<mailto:metsger@cse.ohio-state.edu>>

Contents

1	Change Log	3
2	To Do	3
3	Introduction	4
4	How to Use This Reference	4
4.1	The Index and Table of Contents . . .	4
4.2	Conventions	4
5	Beginning to Program	5
5.1	Java Conventions	5
5.2	Writing a Program	5
5.3	Compiling and Running Programs . . .	5
5.4	Compilation Errors	5
6	Basic Types and Variables	6
7	Operators and Expressions	7
7.1	Mathematical Operators	7
7.2	Boolean Operators	7
8	Using the SavitchIn API	8
9	String Methods	9
10	Flow Control	10
10.1	If Statements	10
10.2	While Loops	10
11	Writing Methods	11
11.1	Method Headers	11
11.2	Parameters	11
11.3	Passing by Value	11
12	Input and Output (Without SavitchIn)	12
12.1	The import Keyword	12
12.2	BufferedReader	12
12.3	PrintWriter	13
13	Exception Handling	14
14	Classes	15
14.1	The Purpose of Classes	15
14.2	Strings	16
14.3	SavitchIn	16
14.4	Static vs. Instance	16
14.5	Constructors	18
14.6	Encapsulation	19
14.7	Program Correctness	20
A	Code Samples	21
A.1	Hello, World!	22
A.2	Making Change from a Vending Machine	23
A.3	Opening and Closing a File	25
A.4	Creating A File	26
B	Glossary	27

1 Change Log

- 24 January 2004 : First version introduced.
- 5 February 2004 : Additions to possible compilation errors and discussion of methods.
- 1 March 2004 : Additional information about I/O and exception handling (courtesy Tim Weale: <mailto:weale@cis.ohio-state.edu>).
- 1 April 2004 : New Code Sample (Hello, World!).
- 2 April 2004 : New Code Sample (ChangeMaker).
- 5 April 2004 : Formatting updates and minor revisions.
- 22 April 2004 : Formatting updates and minor revisions.
- 23 April 2004 : Flow Control Section added, Document contents revised to reflect better logical flow.
- 27 June 2004 : Glossary added, edits for content.
- 28 June 2004 : Brief discussion of classes.

2 To Do

- Add more code examples, especially for arrays and classes.

3 Introduction

This document is provided for students taking CIS 201 at the Ohio State University. It provides a rudimentary description of pertinent elements of the Java programming language. Its goal is to provide a quick reference guide for students in the course who are having trouble finding answers in other resources.

In this document, you will find information on program writing and compilation, common syntax (compilation) errors, the SavitchIn API, mathematical and Boolean operators, and basic String methods. These categories cover most of the material for the first few weeks and should provide enough information to help students complete homeworks and lab assignments.

This document is not intended as a replacement for information contained in the book; it is only a quick-reference guide and does not have detail necessary to do well in the course.

4 How to Use This Reference

4.1 The Index and Table of Contents

This reference comes with a table of contents and an index. If you use them well, you'll have little trouble finding the information that you're looking for quickly and easily. Each of the topics and functions has its own index entry.

4.2 Conventions

Commands and programming language constructs are indicated by a different font, size, and color, like this:

```
javac <filename>.java
```

When commands are printed, they will have a variety of characters that may not be familiar. For example, the angle brackets (< and >) are used to indicate that whatever comes between them must be filled in by the user. A table summarizing these special characters is below:

Character	Meaning	Example
< >	Required value from user.	javac <programName>.java
[]	Optional value from user.	java [-verbose] <programName>

Neither of the special characters should be replicated when you type the command. For example, no angle brackets are used here:

```
javac printMyName.java
```

Pertinent vocabulary words are *italicized*. In select cases, concepts or examples are highlighted with the use of a sans serif font.

5 Beginning to Program

5.1 Java Conventions

Java is a *case-sensitive* language. That means that `printMyName` is different than `PRINTMYNAME`. When you save your files, then, you *must* use exactly the same case as you did when you named your program.

When you name your program or declare variables, you are not allowed to start the name with a number. Thus `1stProgram` is an *invalid* name for a program, but `programNo1` is perfectly fine.

5.2 Writing a Program

Every single Java program begins with the same first two lines and ends with the same last two lines:

```
public class <programName> {
    public static void main (String[] args) {
        // code goes here
    }
}
```

The first two lines tell Java that you are beginning a program. Your file must be saved in `<programName>.java`. So if you call your program `printMyName`, you must save the source code file in `printMyName.java`.

The opening and closing braces (`{` and `}`) open and close a code block. These are also used for **if** statements and **while** loops.

5.3 Compiling and Running Programs

To Compile
To Run

```
javac <programName>.java
java <programName>
```

You can also compile and run programs inside of XEmacs. Use the **JDE** menu to do this.

5.4 Compilation Errors

When you compile your programs, you will often run into different kinds of errors. This is a very brief list that will attempt to explain a couple common errors. Note that Java is nice enough to tell you which line of code the error occurs on.

Error	Meaning	Resolution
<code>Class <programName> is public, should be declared in <programName>.java</code>	This means that your <code><programName></code> is not the same as your filename.	Rename the file to match the name of your program, or rename the class to match the filename. The latter is easiest.

Error	Meaning	Resolution
Missing Semicolon	You have written a statement that should be terminated with a semicolon, but is not.	Add a semicolon (;) at the end of the statement. Any statement that can be used to open a code block (e.g., an if or while) is not terminated by a semi-colon.
Not a Statement	You have probably tried to declare a variable that starts with an invalid character.	Change the variable name so that it starts with a letter and contains no mathematical or boolean operators.
Undefined Symbol	This means that a variable you have declared or a program component you have referenced—like SavitchIn—is unavailable.	Make sure you have declared your variables. If you get this error when you use a SavitchIn method, you need to make sure that the directory that contains <programName>.java also contains SavitchIn.java.

Errors can also occur during the execution of the program (these are called *run time* errors). These occur when a method is misused (if, for instance, you attempt to use `charAt` on an index that does not exist) or when there is a mathematical error (*e.g.*, a number is divided by zero). Another type of error (*logic error*) occurs when you have made a logical mistake. The program may execute without errors, but the results are unexpected.

6 Basic Types and Variables

These are the basic types you need to be familiar with and the kind of values they can represent:

Type	Value
int	{ ..., -1, 0, 1, ... }
char	any single character
double	real numbers (64 bits of precision)
float	real numbers (32 bits of precision)
boolean	true, false
String	any group of characters (including single characters)

A *variable* can be thought of as a container that holds data. Each variable has a *type*, as outlined in the list above. The type of a variable designates what sort of values (data) a variable can hold. To extend our analogy of containers, think of the difference between a vase and a gas can—gas never goes into a vase, and flowers never go into a gas can (unless, of course, we’re practicing some kind of postmodern art). In the same way, a String variable holds *only* Strings, never integers.¹

There are other types that are considered *complex types*. Strings, for instance, are complex types, because we can invoke methods on them. See *String Methods on page 9*.

7 Operators and Expressions

An *operator* is a symbol in Java that performs an action. There are two primary types: *mathematical* and *Boolean*. Each operator can also be used in an *expression*. An expression is a Java statement that uses those operators. When evaluated, expressions return values.

7.1 Mathematical Operators

In mathematics, these operators are the symbols you have become used to throughout your time in school:

Operator Meaning

+	addition
-	subtraction
*	multiplication
/	division
%	modulus

Of all of these operators, the modulus operator is usually the most confusing. It returns the remainder when two numbers are divided:

```
10 % 5 = 0
12 % 5 = 2
137 % 10 = 7
```

When in doubt of the result of evaluating an expression containing a modulus operator, simply do long division. The remainder when you perform long division is the result of applying the modulus operator.

7.2 Boolean Operators

Boolean operators return a true or false value, not a number. Some of these could also be listed in the math operators section because they have mathematical meanings, too, but weren’t because of how they are evaluated.

Operator Meaning

&&	and (evaluates to <i>true</i> when every one of the parts of the expression is <i>true</i>)
	or (evaluates to <i>false</i> when every one of the parts of the expression is <i>false</i>)

¹Of course, a String variable may hold the *representation* of an integer. For example, `String aNumber = ‘‘1234’’` is a valid statement; but we cannot perform mathematical operations on that String (*e.g.*, `aNumber + 2`).

!	not (evaluates to <i>true</i> when the argument is <i>false</i>)
==	equal (evaluates to <i>true</i> when both arguments are “the same”—though exactly what this means changes for each variable type)
!=	not equal (evaluates to <i>true</i> when both arguments are not “the same”)
>	greater than (evaluates to <i>true</i> when the left-hand-side is greater than the right-hand-side)
>=	greater than or equal to (evaluates to <i>true</i> when the left-hand-side is greater than or equal to the right-hand-side)
<	less than (evaluates to <i>true</i> when the left-hand-side is less than the right-hand-side)
<=	less than or equal to (evaluates to <i>true</i> when the left-hand-side is less than or equal to the right-hand-side)

Boolean expressions are named for the mathematician George Boole , who invented Boolean logic. See <http://www-gap.dcs.st-and.ac.uk/~history/Mathematicians/Boole.html> for more information about this mathematician.

Boolean expressions are used to control the way that programs are executed. This is known as *flow control*. For more information, see the section called *Flow Control* on page 10.

8 Using the SavitchIn API

This section describes several of the methods provided by Walter Savitch (the author of our textbook). He wrote them to facilitate keyboard input from users. The normal methods used by Java for getting user input are not intuitive (though once you acquire some familiarity with them and their uses, you will find them easier to use than their C++ counterparts).

Method	Use	Example
<code>readLine ()</code>	Reads a String from the user.	<pre>String test = SavitchIn.readLine ();</pre>
<code>readLineInt ()</code>	Reads an integer (int) from the user.	<pre>int test = SavitchIn.readLineInt ();</pre>
<code>readLineDouble ()</code>	Reads a decimal number from the user.	<pre>double test = SavitchIn.readLineDouble ();</pre>
<code>readLineNonwhiteChar ()</code>	Reads a character (char) from the user, ignoring any whitespace and newlines.	<pre>char test = SavitchIn.readLineNonwhiteChar();</pre>

A more detailed reference can be found at

<<http://www.cis.ohio-state.edu/cis201/reference/SavitchIn/index.html>>.

These methods are all used to read input from the keyboard. They have been written by the author of our textbook (Walter Savitch) to make introductory input easier on students (believe it or not). There are several other methods which we will not use, but interested students can find information at the URL above.

This bit of software is built on top of Java's native methods for getting input and output. You can read more about them below, in the section titled *Input and Output without Using SavitchIn*.

9 String Methods

This section will introduce you to a number of the String methods and describe them in some detail. Its goal is to serve as a quick reference to those methods that you need to be familiar with. There are many more of them, which you can find at

<http://java.sun.com/j2se/1.4.2/docs/api/java/lang/String.html>.

Method	Description	Return Type	Example
<code>charAt (int pos)</code>	Returns the character at a specific index of a String; if the index does not exist, program dies.	char	<pre>char c = s1.charAt (0);</pre>
<code>equals (String s)</code>	Tells whether or not two strings are the same. This is case-sensitive!	boolean	<pre>boolean b = s1.equals (s2);</pre>
<code>indexOf (String s)</code>	Returns the index of the first occurrence of s in the String; if s does not occur, returns -1.	int	<pre>int i = s1.indexOf ('Lord');</pre>
<code>substring (int pos1, int pos2)</code>	Returns the String between positions pos1 and pos2 - 1; if pos1 < pos2, or either one does not exist, program dies.	String	<pre>String t = s.substring (10, 13);</pre>
<code>length ()</code>	Returns the number of characters in the String.	int	<pre>int i = s.length ();</pre>

These methods are all used to get information about Strings.

The idea of methods is a little complicated, but this analogy may help you understand them a little better. A *method* is a question that we ask of an object—in this case, the object is a String. We store the answer to that question in a variable or, if the answer is true or false, use it in a Boolean expression.

Think of a complex type as a sentient being whose function is to store information and answer questions about that information. The phrase `int i = s.length ();` might be interpreted as the question, “s, what is your length?” You can similarly cast every other String method the same way.

10 Flow Control

Flow control is a term that describes the way that code executes. Code may execute in a linear fashion, it may branch², or it may be executed repeatedly. We are already familiar with executing code in a linear fashion—any code that we’ve seen until this point has been linearly executed. Now we’ll introduce the *if* statement and *while* loop.

10.1 If Statements

If statements are used by Java to cause code to branch. The syntax is as follows:

```
if (<condition>) {
    // execute this code if <condition> is true
}
else {
    // execute this code if <condition> is false
}
```

You will also see a variation on the standard if-else statement called the *if-else if-else* statement, which will handle an arbitrary number of conditions. The syntax for this looks like this:

```
if (<condition>) {
    // execute this code if <condition> is true
}
else if (<condition2>) {
    // execute this code if <condition2> is true, but <condition1> is false
}
else {
    // execute this code if both conditions are false
}
```

Though we’ve only shown two conditions being checked here, you can check as many as you’d like.

10.2 While Loops

While loops are programming constructs that allow us to repeatedly execute code (this is called *iteration*). The syntax for a while loop looks like this:

```
while (<condition>) {
```

²Given some condition, a certain branch of code may execute. If that condition is false, then another branch is taken.

```
// code goes here
}
```

As long as `<condition>` is true, the code will execute. This means that the variable to which `<condition>` is bound *must* change each time the loop executes; otherwise it will run forever!

You can think of a while loop as an if-statement that is executed over and over until the Boolean expression (the `<condition>` in the parentheses) evaluates to false. Just remember that there are no else-blocks in a while loop!

11 Writing Methods

Methods are used in Java for two principle purposes: program organization and easy reuse of code. Methods are composed of a method signature (or header) and a method body. A method body is simply a code segment—you've been writing these all quarter.

And without realizing it, you have been writing method headers all quarter:

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

is a method header.

11.1 Method Headers

Method headers have the following syntax:

```
public static <returnType> <methodName> (<type> <paramName>, ...)
```

The `<returnType>` can be any type available to Java. If you wish to specify that you are not returning any value (a common occurrence when you use a method to output information), `<returnType>` is **void**.

The name of the method should be descriptive; when naming it, make sure to convey precisely what the method does without making the name too long. The conventions applied to naming variables also apply to methods; you are not allowed to start a method name with a number.

11.2 Parameters

Parameters are variables that can be used in the method body. You can make use of any parameter in a method in any legal way that you would like. You should consider that a parameter is simply renaming a variable from another method and using it in the one you're writing.

11.3 Passing by Value

There's a catch to this, though. If a parameter is a primitive type (like an `int`, for example), it cannot be changed within the method. This is because it is passed by value. So a method that looks like this:

```
public static void testMethod (int changeMe) {
    changeMe = 5;
}
```

doesn't actually change the value of the integer `changeMe`. If you write a program that uses this method like this:

```
public static void main (String[] args) {
    int willThisChange = 4;
    System.out.println ('willThisChange before testMethod (): ' + willThisChange);

    testMethod (willThisChange);
    System.out.println ('willThisChange after testMethod (): ' + willThisChange);
}
```

The output looks like this:

```
willThisChange before testMethod (): 4
willThisChange after testMethod (): 4
```

So even though we changed the value of `willThisChange` (by setting the value of `changeMe`), it remains the same. That is because parameters that are passed by value are *copied into* the method.

12 Input and Output (Without SavitchIn)

To this point, all of our input has been received from the keyboard, and we have used Walter Savitch's *utility class* `SavitchIn` to do this for us. Unfortunately, this does not allow us to get information any other way—we're restricted to using the keyboard for input. But what if we want to get input from a file? We'll need something new.

It turns out that Java uses the same structures to get input from the keyboard as it does to get input from a file. We'll examine the two in tandem.

12.1 The `import` Keyword

Throughout the course, we have been able to use `SavitchIn` because it was in the same directory as the code that we wrote. This time, we will make use of Java classes that are not available inside of our directory and are not loaded by default (like `String` is). This requires that we use the `import` keyword. Simply include this line in your code to gain access to all of the necessary functions and classes:

```
import java.io.*;
```

This line goes at the very top of your program, before `public class <programName>` and the main method.

12.2 `BufferedReader`

`BufferedReader`s are used to read input from a file or the keyboard. Following is a list of three ways to declare `BufferedReader`s.

12.2.1 Syntax

To read from the file "in.txt", use this syntax:

```
BufferedReader br = new BufferedReader (new FileReader (“in.txt”));
```

To read from a file whose name is given as input by the user:

```
String str = SavitchIn.readLine ();  
BufferedReader br = new BufferedReader (new FileReader (str));
```

To read from the keyboard:

```
BufferedReader br = new BufferedReader (new InputStreamReader (System.in));
```

12.2.2 Methods

To read from a `BufferedReader`, we use the `readLine()` method—just like you used to use for `SavitchIn`. `readLine()` returns the next line of input (e.g., the next line of the file) or **null** if there is no input left to read (e.g., if you have reached the end of the file).

The following will read a complete file from `BufferedReader` input:

```
String input = br.readLine ();  
while(input != null) {  
    // do something with the string here!  
    input = br.readLine ();  
}
```

Note that `readLine()` *always* reads `String` values; it cannot return an `int` or `double`. These values, then, must be converted once they are read. To do this, you need to use the `Integer` and `Double` utility classes provided by Java.

To close a `BufferedReader`, use the `close()` method:

```
br.close();
```

Closing streams is important! (More so when writing than reading; but even when reading.)

12.2.3 Converting Strings to Other Types

Target Type	Method Used	Example
<code>int</code>	<code>Integer.parseInt(String)</code>	<pre>int i = Integer.parseInt (“42”);</pre>
<code>double</code>	<code>Double.parseDouble (String)</code>	<pre>double d = Double.parseDouble (“3.14159”);</pre>

Any `String` methods still apply, so if you need to get a character out of the `String`, you can use `charAt()`.

12.3 PrintWriter

A `PrintWriter` is used to write to a file (or any other output stream; in this class, only files and your monitor are considered). Following is the syntax to declare `PrintWriters`.

12.3.1 Syntax

To write to the file “out.txt”:

```
PrintWriter pw = new PrintWriter (new FileOutputStream (“out.txt”));
```

To write to a user-specified file:

```
String str = SavitchIn.readLine();  
PrintWriter pw = new PrintWriter (new FileOutputStream (str));
```

Instead of using a FileOutputStream object, you can also use a FileWriter, like this:

```
PrintWriter pw = new PrintWriter (new FileWriter (“out.txt”));
```

12.3.2 Methods

The same methods are used to output to a file as are used to output to your screen. (Remember System.out.println()?)

The print() method is used to print information on a single line, without appending a new line:

```
pw.print (“This is on one line”);  
pw.print (“This is still on the same line”);
```

The println() method is used to print information on multiple lines:

```
pw.println (“This is on one line”);  
pw.println (“This is on another line”);
```

And, like the BufferedReader, there is a method to close the stream:

```
pw.close();
```

Note that if you do not close the file using this method, the contents of the file may not be written! If you wind up having an empty output file, then you probably need to make sure that you are closing the PrintWriter object that created it.

13 Exception Handling

Exception handling is Java’s means of recovering gracefully from an error. It accomplishes this by means of a try-catch block, which looks like this:

```
// Initialize anything you need here!  
try {  
    // Create your BufferedReaders and PrintWriters here  
    // Use them here  
    // Close them here  
}
```

```

catch (FileNotFoundException e) {
    // Handle the case of a FileNotFoundException here
    // Typically, print an error message and exit
    // Usually only done with a BufferedReader
    // PrintWriters create the file if it's not found.
}

catch (IOException e) {
    // Handle a generic IOException here
    // Typically, print an error message and exit.
}

// Additional Code goes here!

```

Exceptions are errors that a method or component “throws,” which can in turn be “caught” by your program. Intuitively, what happens is that Java “tries” to execute the try block; if it finds an error, then it “catches” it and performs the code written in the catch block.

Note that you can use multiple catch statements for each try; this allows you to catch multiple kinds of errors, as shown above.

Exceptions allow a Java program to recover from errors that would normally cause the program to crash, *e.g.*, a divide by zero error. Programmers typically use exception handling to either output or create error messages for use in program, but mission-critical applications will also use exceptions to increase stability.

14 Classes

Complex data types in Java are known as *classes*. A class is considered complex for a variety of reasons, and we’ll only provide an introduction to them here.

The first example of classes we should examine are Strings, as they provide a good example from a *client perspective* of what classes are like. By client perspective, we mean that we are *clients of* (or using) Strings, not writing them. The reason we emphasize this now is because our perspective is changing—soon we will be writing classes, and we will have an *implementor perspective*.

14.1 The Purpose of Classes

Classes are used to model *objects* that we encounter in real life. These objects may be abstract (like a String, for instance) or concrete (like a ball). While the terms *object* and *class* are used interchangeably for the most part, a distinction is made between the two in this summary. An *object* is the real-life entity that we wish to model. The actual model of that object is called a *class*. Nonetheless, an instance of a class is often called an *object* in programming parlance. We will refer to them as *instances*. (The definition of this term is found below.)

We maintain this distinction mostly for clarity, and because we often find that our models of objects do not necessarily correspond exactly to the objects they model. This discrepancy usually occurs because we do not need to explicitly model *everything* about an object. (If we are classifying fruit, for instance, we might only want to distinguish them from each other by color and size—the two characteristics most readily identifiable from a distance.)

The easiest objects to model are often mathematical concepts. It’s very easy, for instance, to model Euclidean points. See Appendix A for some examples.

14.2 Strings

We first examine Strings because they are a complex type—a class. We learned early on that individual Strings may be queried by methods. For instance, we can ask a string `s1` how many characters it has by using the `length()` method: `s1.length()`.

Primitive types, however, do not admit methods. There are no methods that we can call on an integer, for instance, or a character. This suffices to show us one of the principle differences between classes and primitive types: any class type may include methods. The behavior of these methods is usually published in what is known as an *Application Programming Interface*, or *API*. Typically the method header is exposed.

These methods are always called by the format `<variableName>.methodName([arguments])`.

14.2.1 Creating Strings Using `new`

For the entire quarter, we have been creating Strings by using code that looks like this: `String s1 = "Hello, World!"` But there is another way to create Strings:

```
String s1 = new String ("Hello World!")
```

This should look a little familiar from our experience with arrays. We use the `new` keyword to allocate a space in memory for the new String, just like we allocated space for our arrays. When we use the `new` keyword this way, we say that we are creating a *new instance of a String*. (We often shorten this to simply say that we are creating a new String; but when we talk about using the `new` keyword in general, we need to use the term *instance*.)

What makes two instances different from each other? In the case of Strings, we can always say that two instances are different because they have different representations. “Hello” is not the same as “World!”. We can also say that they are different because using the `equals()` method will return `false`.

14.2.2 Creating New Instances Using `new`

In general, you can create a new instance of a class by the following syntax:

```
<className> <variableName> = new <className> ()
```

The parentheses should make us think of a method. You’ll notice when we created a new String, we supplied a String between the two parentheses as an argument. Tuck this away in the back of your head; we’ll return to it in a little while.

14.3 SavitchIn

We have also seen another kind of class that has a different format for calling methods: `SavitchIn`. `SavitchIn` is a *utility class*, that is, a class that contains only public static methods. (These are the kind we’ve been writing during throughout the quarter.) These methods are called with the format `<className>.methodName([arguments])`, such as `SavitchIn.readLine()`.

14.4 Static vs. Instance

This raises a somewhat difficult concept. Throughout the quarter, we have been writing *static methods*, that is, methods that are declared with the `static` keyword. What this means, exactly, is difficult to explain. In the easiest sense, we can think of a static method or variable as one that is shared.

But shared between what?

Between individual *instances* of that class. When we create a new String, we create a new *instance* of a string. The characteristic of an instance is that it is different from every other instance. For example, one instance of a String is “hello” and another is “world”.

But what would it mean to have two different instances of a SavitchIn object? The answer is *nothing*. We are certainly allowed to do the following:

```
SavitchIn si = new SavitchIn ();
           :
si.readLine ();
```

But the action of the instance `si` is exactly the same as any other instance of SavitchIn. This is the effect of using the static keyword in front of a method. This somewhat lengthier example may explain this idea a little more.

```
public class SDemo1 {
    public static int x = 0;
}

public class TestSDemo1 {
    public static void main (String [] args) {
        SDemo1 sd1 = new SDemo1 ();
        SDemo2 sd2 = new SDemo1 ();

        sd1.x = sd1.x + 1;
        System.out.println (sd2.x);
    }
}
```

The question we ask now is this: what does the line `System.out.println (sd2.x);` actually output? The answer is `1`. This is because the variable `x` is *shared* between the two instances of SDemo1. The `x` that `sd2` owns and the `x` `sd1` owns are exactly the same `x`. They are stored in the same memory location.

We call `x` a *static variable* because it is prefixed with the `static` keyword. In the class SavitchIn, `readLine()` is a static method because it is prefixed with the keyword `static`. Global class variables (the kind that are not declared inside of a method) that are declared static may not be accessed inside of an instance method; nor may instance variables be accessed inside of a static method.

The opposite of static is *instance*. We declare instance methods and variables by leaving off the static keyword. A similar example may help to illustrate the difference between static and instance.

```
public class IDemo1 {
    public int x;
}

public class TestIDemo1 {
    public static void main (String [] args) {
        IDemo1 id1 = new IDemo1 ();
    }
}
```

```

    IDemo1 id2 = new IDemo1 ();

    id1.x = 1;
    id2.x = 2;
    System.out.println (id1.x + " " + id2.x);
}
}

```

When we perform output here, we'll see that `id1` and `id2` have two different values for `x`. This is because `x` is not a shared (or static) variable. When we create new instance of the class `IDemo1`, we are creating two separate locations for `x` to “live.” One is reserved for `id1` and the other for `id2`.

These concepts are fairly difficult to understand and may require some additional review. The takehome message for the moment is that we access instance variables and methods using the instance's name (like `Strings`) and static variables and objects using the class's name (like `SavitchIn`).

14.5 Constructors

When we began to discuss creating variables, we noted that we could supply an *initial value* to a variable. There is an analogue to this in complex types known as the *constructor*. A constructor is an instance method that has the same name as the class to which it belongs. For instance:

```

public class IDemo1 {
    private int x;
    public IDemo1 () {
        x = 0;
    }
}

```

The method `IDemo1()` takes no parameters and sets the value of `x` to zero. It is a constructor *because it has the same name as the class*. Constructors do not have any return type associated with them. See 14.6 on the following page for more information about what the `private` keyword means.

We may also add parameters to a constructor in order to allow programmers to specify pertinent initial values for instances of a class:

```

    :
public IDemo1 () {
    x = 0;
}

public IDemo1 (int _x) {
    x = _x;
}
    :

```

In this case, we have two constructors; one which takes no parameters, and one which takes a single integer. Constructors which take no parameters are called *default constructors*. The second constructor takes the supplied value and sets the instance variable `x` to it.

These two constructors may be used as follows:

```
⋮
IDemo1 id1 = new IDemo1 (); // default constructor
IDemo1 id2 = new IDemo1 (5); // provides initial value
⋮
```

The first line uses the default constructor to set the instance variable `x` equal to zero. The second line sets the instance variable `x` to 5.

14.6 Encapsulation

We have been dancing around the idea for the past couple sections that we need to be able to access information about an instance of a class. For instance, many of our programs have relied upon the length of a `String`—so we use a method called `length()` to query a particular `String` about how many characters are in it.

The process of defining how clients of our classes may access and change data in instances of the class is called *encapsulation*. This is a mouthful! The colloquial definition of “encapsulate” is to put something into a capsule or container. Most of the time, people just say, “I’d like to put `x` into `y`” rather than using the more highbrow “encapsulate.” And we might well question its use here, as well; haven’t we already defined *assignment* as putting a value into a container?³ It turns out that classes require a somewhat more complicated word (and definition) because they can hold several different *types* of data.

We accomplish encapsulation by using *get* and *set methods*. (Examples of these may be found in the Appendix.)

14.6.1 Get Methods

Get methods are sometimes also known as *accessor methods* because they access the variables contained in a class. Get methods are fairly boring and usually consist of a single line:

```
return <variableName>;
```

The goal is to return a value stored within a class, so a get method will *always* have a return type (this type will be the same as the variable you are returning).

14.6.2 Set Methods

Set methods (also known as *modifier methods*) are a little more interesting.

Our motivation for having set methods is to prevent a client of our class from making arbitrary modifications to data fields (that is, the variables) in a class. For instance, maybe you are writing a piece of software that another programmer will use to measure time-related data. In certain applications, it wouldn’t make sense to allow the programmer to use negative time. Another example is that you are writing a class that will hold information about people for a phonebook—it wouldn’t make sense to allow entries in the phonebook with missing names.

Most of the time, set methods do *not* have return types.

³The colloquial way of defining assignment is putting data into a variable.

14.6.3 `private` vs. `public`

These two kinds of methods may leave us wondering a little bit about why they are necessary. We have seen that set methods will allow us to tell programmers how they may modify data in our class, but what prevents them from accessing that data directly? (If you look at the program `SDemo1` on page 17, you'll notice that we directly modified `x` by using `SDemo1.x`. The question I am asking now is, "What prevents us from doing this with *any* class or instance?")

The answer is that we can modify both variables and methods using the `private` keyword. This restricts their use to the class that contains those variables in methods. This forces programmers to use the get and set methods to access and modify the variables that are contained in these classes.

14.7 Program Correctness

The motivation for both Constructors and Encapsulation is to ensure program correctness. If programmers cannot rely on instances of classes that you create to have a well-defined initial value and well-defined behavior, how will they be able to use them with confidence? And how will they be able to go to their bosses and say, "This program is proven to work?"

While we are introducing a large level of obfuscation here, we are also introducing a very significant and very important advance in our programming vocabulary. We made a small leap when we went from Flow Control to Methods, and we are making another leap (a bit larger this time) by moving on to Classes. Classes allow us to express very complicated ideas in programming; this is why we need to introduce constructs (like get and set methods, or constructors) to manage how these ideas are expressed and changed throughout programs.

A Code Samples

A.1 Hello, World!

A.2 ChangeMaker

A.3 Opening and Closing a File

A.4 Creating a File

A.1 Hello, World!

The first program ever written in any program language is the Hello, World! program. It is designed to acquaint the reader with how to construct elementary programs and do output to the screen.

```
/* file : HelloWorld.java
 *
 * Ethan Metsger / CIS201 / 29 March 2004
 *
 * This illustrates the basic HelloWorld application, which simply prints
 * out a message to the screen. We are using this program to provide
 * some basic introduction to the concept of programming.
 *
 * This, for instance, is a multi-line comment, and is ignored by the
 * Java compiler. */

public class HelloWorld {
    /* The phrase "public class" is what opens up a program. What
     * follows is the program name. In general, to write a program, you
     * always begin it with public class <ProgramName>, where
     * <ProgramName> is a name that you choose. */

    public static void main (String [] args) {
        /* We don't need to understand this line in any detail right now.
         * At a high level, this is Java for "I'm starting to write the
         * guts of my program." */

        System.out.println ("Hello, World!");
        // To output messages to the screen, we use System.out.println().

        // The double-slash, by the way, is a single-line comment.

        /* Every statement in Java is terminated by a semi-colon. It's a
         * bit like a period or question mark in English. */

        /* Any String value (a String is just a collection of
         * characters) is put into quotation marks ("). */
    } // every opening brace ({) must have a matching closing brace (})
}
```

A.2 Making Change from a Vending Machine

This program is designed to acquaint the reader with mathematical operators and expressions, especially the less familiar modulus operator.

```
/* file : ChangeMaker.java
 *
 * Ethan Metsger / CIS 201 / 2 April 2004
 *
 * The change maker demonstrates the use of mathematical expressions and
 * operators. The idea is that we provide the program an input of some
 * number of cents and it returns the amount of change we should receive
 * back, separating it into the right number of quarters, dimes, nickels,
 * et c. */

public class ChangeMaker {

    public static void main (String[] args) {

        // returned change
        int quarters, dimes, nickels, pennies, remaining;

        // how much money the user provided
        int input;

        /* We'd like to fix the cost at 85 cents for now. We use the keyword
         * "final" to denote that we cannot change this value later in the
         * program.
         *
         * As an exercise, let the user specify the cost. */
        final int cost = 85;

        // get user input
        System.out.print ("The cost of the chips is " cost ". " +
            "How much money did you put into the vending " +
            "machine? ");

        /* Since we're reading an integer, we use the SavitchIn.readLineInt()
         * method. */
        input = SavitchIn.readLineInt ();

        /* The first step in our algorithm is to determine how much change
         * we need to return. We do this by subtracting the cost from the
         * input.
         *
         * Food for thought : what should happen if the change to return is
         * negative?? */
        remaining = input - cost;
```

```

/* Now we need to calculate how many quarters, et c., to return.
 * This is done as follows:
 *
 * Divide the remaining amount by the cent-value of the coin in
 * question. In this case, if the coin is a quarter, we need to
 * divide the remaining change by 25 cents. This tells us how many
 * quarters fit into our remaining change.
 *
 * We then need to know how much change is left over. We can do this
 * in a couple ways, but the best way is to use the modulus operator.
 * If the coin in question is a quarter, the remaining change is
 * the current remaining change mod 25. */
quarters = remaining / 25;
remaining = remaining % 25;
dimes = remaining / 10;
remaining = remaining % 10;
nickels = remaining / 5;
remaining = remaining % 5;
pennies = remaining;
remaining = 0; // this line is not necessary

// output end values
System.out.println ("You received " + quarters + " quarters, " +
                    dimes + " dimes, " + nickels + " nickels, " +
                    "and " + pennies + " pennies.");

} // end program

} // finish program

```

A.3 Opening and Closing a File

This program demonstrates the use of BufferedReaders and Exception handling.

```
/* This program gets a filename from a user, and attempts to open and
   read the first line of the file.  If the file doesn't exist, then the
   user must re-attempt to enter a filename.  */

import java.io.*;

public class fileLoop {
    public static void main(String [] args) {
        boolean success = false;
        while( !success ) {

            try {
                // Try to open the file and read the first line of the file.
                System.out.print("Please enter the filename: ");

                // Read from the keyboard
                BufferedReader kinput = new BufferedReader (new
                    InputStreamReader (System.in));
                String str = kinput.readLine();

                // Open and read from the file
                BufferedReader br = new BufferedReader (new FileReader (str));
                String input = br.readLine();

                kinput.close();
                br.close();
                success = true;
            }

            catch (FileNotFoundException e) { // Catch an exception!
                System.out.print("File doesn't exist.  Please enter another'' +
                    ''filename:");
            }

            catch (IOException e) { // Catch another exception!
                System.out.println("ERROR! IOEXCEPTION!");
                System.exit(1); // Major problem!  Exit!
            }
        } //end: while

        System.out.println("File opened and closed successfully!''');
    }
} // end: fileLoop
```

A.4 Creating A File

This program demonstrates the use of Exception Handling and PrintWriters.

```
/* This program takes user input from a keyboard and writes it directly
   to a file.  If the user enters "EXIT", the program finishes.*/

import java.io.*;

public class fileWrite {

    public static void main(String [] args) {

        String str;

        try {
            // Read from the keyboard
            BufferedReader kinput = new BufferedReader (new InputStreamReader
                (System.in));

            str = kinput.readLine();

            // Open and write to the file
            PrintWriter pw = new PrintWriter (new FileOutputStream
                ("out.txt"));

            while (!str.equals ("EXIT")) {
                br.println(str);
                str=kinput.readLine();
            }

            kinput.close();
            pw.close();
        }

        catch (IOException e) {
            System.out.println("ERROR! IOEXCEPTION!");
            System.exit(1);
        }

    }

}

} //end: fileWrite
```

B Glossary

Establishing a good vocabulary is essential in any field, particularly in technical fields like Computer Science. Communication on a basic level is difficult without a common vocabulary. These particular terms are key to understanding fundamental aspects of programming.

algorithm A set of instructions, like the recipe to bake a cake or make change from a vending machine.

application programming interface Or API. This is a list of the public members of a class, so that programmers will be able to call methods and access any public data fields of the class.

array A collection of elements that are all the same type; an array of characters, for instance, is like a String. The length of an array may be accessed using `<arrayName>.length`.

assembler This is the program that translates *assembly language* instructions into *machine code*. An intermediate step some *compilers* take in translating *source code* is to translate it into assembly language first and let the system's assembler translate them into machine code.

Boolean condition A condition that is either true or false. These are named for the mathematician George Boole.

byte code A translation of Java source code into a universal format understood by all *Java Virtual Machines*. Byte-code allows you to write a program one time that will run anywhere—in other words, it ensures true portability between different processors (or architectures).

class A complex type in Java. Classes differ from primitive types in that they can be queried by *methods* and may contain several different types of data.

compiler This is the program that takes *source code* and translates it into *machine code*. The Java compiler translates source code into *byte code*.

constructor A constructor is a method for a class that provides new *instances* of the class with an initial value.

encapsulation Colloquially, a fancy word for putting something into something else. In programming terms, we think of a class as a “capsule” that holds data (a bit like a variable, but somewhat more complicated). The term *encapsulation* refers to how we access and change the information inside of a class.

exception An exception is an error that occurs during program execution. When an exception arises, it is said to be *thrown*. Exceptions can also be *caught*, so that Java can recover gracefully.

flow control The combination of *iteration* and *selection* is called flow control. Flow control describes in what order and whether pieces of a program are executed.

hardware The physical components of your computer, like your RAM, CPU, and peripherals (monitor, mouse, printer, et c.).

infinite regress Also sometimes referred to as the bootstrapping problem. An example of infinite regress is asking the question, “What compiles the compiler?” or “What program loads the operating system?”

instance (1) *n.*, An instance of a class, declared using the `new` keyword; (2) *adj.*, a type of variable or method; these variables and methods lack the `static` keyword.

interpreter A program that translates *source code* into *machine code* as it reads it. Interpreters are slower than compilers because the program only executes through this proxy. Interpreters tend to be more portable, but incur performance penalties.

iteration Repeatedly executing a set of instructions until the *Boolean condition* under which those instructions are executed is false.

Java Virtual Machine Or JVM. This is a program that resides on your computer that translates *byte code* into the *machine code* associated with your processor. It is an *interpreter* for Java.

machine code These are the ones and zeros that make the instructions that your computer understands.

method A self-contained body of code responsible for performing an action. Colloquially, we can think of a method as a bit of code that answers a question. The advantage of methods is that they are reusable (*e.g.*, in another program) and easier to debug.

operating system The program responsible for managing how *software* interacts with the *hardware*. Examples are WindowsXP, Linux, other brands of UNIX, et c.

operator A symbol in the Java language that manipulates data and returns a result. The two kinds of operators in Java are mathematical operators (+, -, *, /, %) and Boolean (&&, ||, !, ==, <, <=, >, >=, !=).

portable A program is said to be *portable* if we can take its *source code* and compile it on a different kind of processor without having to modify the code (*e.g.*, compiling a program written for an Intel-based processor on a Macintosh).

selection Choosing a set of instructions to execute based on a *Boolean condition*; this is accomplished by using if statements.

source code A usually human-readable translation of an algorithm into a programming language like Java.

software Programs that run on your computer, like WindowsXP or MicroSoft Office (or Linux and OpenOffice.org).

static A keyword in Java that modifies both variables and methods. Its colloquial meaning is “shared.” When used on a variable, static means that each instance of a class shares the same variable. When used on a method, static means that the method should be accessed by `<className>.methodName ([parameters])`.

type In programming terms, a kind of data. Types can be either simple (*primitive*) or complex (*class types*).

utility class A class that contains only static methods.

variable Colloquially, a container that holds data; variables have both a *type* and a name. More precisely, a variable is a reference to data held in the computer’s memory.

Index

BufferedReader, 12

Classes, 15

Converting Strings to other Types, 13

Double.parseDouble, 13

Errors, compilation, 5

Errors, logic, 6

Errors, run-time, 6

Exception Handling, 14

George Boole, 8

Glossary, 27

If statements, 10

if-else if-else Statements, 10

import, 12

Integer.parseInt, 13

Methods, 11

Methods, parameters, 11

Operators, boolean, 7

Operators, definition, 7

Operators, mathematical, 7

PrintWriter, 13

Programs, beginning, 5

Programs, compiling and running, 5

Programs, examples, 21

Programs, Hello, World, 22

SavitchIn API, methods, 8

SavitchIn API, web reference, 9

String Methods, 9

Types, basic, 6

Types, complex, 7

While Loops, 10