Accessible Objected-Oriented Programming Concepts for Blind Students

By:
Richard Baldwin

Accessible Objected-Oriented Programming Concepts for Blind Students

By:
Richard Baldwin

Online:
< http://cnx.org/content/col11349/1.6/ >

C O N N E X I O N S

Rice University, Houston, Texas

Collection structure revised: August 21, 2011
PDF generated: August 21, 2011
For copyright and attribution information for the modules contained in this collection, see p. 47.

Table of Contents

Chapter 1

Getting Started[1]

1.1 Table of Contents

1.2 Preface

1.2.1 General

This module is part of a collection of modules designed to make object-oriented programming concepts accessible to blind students.

[1] This content is available online at <http://cnx.org/content/m40794/1.1/>.

Blind students should not be excluded from computer programming courses because of inaccessible textbooks. Because of its text-based nature, computer programming is fundamentally an accessible technology. However, many textbooks adopt and use high-level integrated development environments with graphical user interfaces that greatly reduce that accessibility.

The modules in this collection present object-oriented programming concepts in a format that blind students can read using tools such as an audio screen reader and an electronic line-by-line Braille display.

In an effort to get and keep the student's interest, these modules make heavy use of programming projects that provide sensory feedback through the use of both sampled sound and MIDI sound.

The collection is intended to supplement but not to replace the textbook in an introductory course in high school or college object-oriented programming.

See http://cnx.org/content/col11349/latest/ [2] for the main page of the collection. If you can see the main page, there is a Table of Contents on the left side of the page. I'm not sure how your screen reader will treat that Table of Contents relative to the other parts of the page.

This module explains how to get started programming in Java in a format that is accessible to blind students.

1.2.2 Prerequisites

In addition to an Internet connection and a browser, you will need the following tools *(as a minimum)* to work through the exercises in these modules:

- An audio screen reader that is compatible with your operating system, such as the NonVisual Desktop Access program (NVDA), which is freely available at http://www.nvda-project.org/ [3] .
- A refreshable Braille display capable of providing a line by line tactile output of information displayed on the computer monitor is recommended (http://www.userite.com/ecampus/lesson1/tools.php [4]).
- The Sun/Oracle Java Development Kit (JDK) *(See http://www.oracle.com/technetwork/java/javase/downloads/index.* [5] *)*
- Documentation for the Sun/Oracle Java Development Kit (JDK) *(See http://download.oracle.com/javase/7/docs/api/* [6] *)*
- A simple IDE or text editor for use in writing Java code.

The minimum prerequisites for understanding the material in these modules include:

- An understanding of algebra.
- An understanding of all of the material covered in the earlier modules in this collection.

1.2.3 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

1.2.3.1 Listings

- Listing 1 (p. 5) . Windows batch file.
- Listing 2 (p. 6) . A test program.

[2] http://cnx.org/content/col11349/latest/
[3] http://www.nvda-project.org/
[4] http://www.userite.com/ecampus/lesson1/tools.php
[5] http://www.oracle.com/technetwork/java/javase/downloads/index.html
[6] http://download.oracle.com/javase/7/docs/api/

1.2.4 Supplemental material

I recommend that you also study the other lessons in my extensive collection of online programming tutorials. You will find a consolidated index at www.DickBaldwin.com [7] .

1.3 Discussion

Considering that during the past fifteen years, I have published several hundred online programming tutorials *(see http://www.dickbaldwin.com/toc.htm* [8] *)*, one might wonder why I am taking the time and expending the effort to publish still another online programming tutorial.

1.3.1 Accessibility is the keyword

When writing and publishing the earlier tutorials, I made no attempt to make them accessible to blind students. Some of them are probably accessible, simply because I used a relatively simple HTML format and didn't include any inaccessible content such as images. However, many of the earlier tutorials make heavy use of images and will therefore be inaccessible to blind students.

In writing this collection of modules *(tutorials)* , I will make a concentrated effort to make them accessible to blind students.

Some of the earlier tutorials that include images do so because the purpose of the tutorial was to teach how to manipulate graphic images.

Other tutorials, however, included images in sample programs simply as a way to provide sensory feedback to the students and give them a feeling of accomplishment. I believe that providing sensory feedback in sample programs makes the process of learning how to program more interesting for the students.

1.3.2 Sound and music

In this collection of modules, I will use sound instead of images to provide sensory feedback.

1.3.2.1 Sampled sound

In some cases, the sound will be of a form that is often referred to as *sampled sound* . In sampled sound, the actual waveform of the sound is sampled as a series of numeric values. At playback time, those numerical values are applied in sequence to a device that reproduces the original waveform and feeds that waveform to amplifiers, speakers, etc. This is the type of sound that is commonly found on a music CD.

1.3.2.2 MIDI sound

In other cases, I will use MIDI sound. Pronounced *middy* , this is an acronym for *musical instrument digital interface*. MIDI is a standard adopted by the electronic music industry for controlling devices, such as synthesizers and sound cards, that emit music. For example, this is the scheme that is usually employed by digital keyboards used in rock bands.

While I am not a musician, I will show you how to write Java code to play some simple melodies. Perhaps after completing this series of modules, you can continue learning on your own to create serious music using MIDI.

[7] http://www.dickbaldwin.com/toc.htm
[8] http://www.dickbaldwin.com/toc.htm

1.4 Writing, compiling, and running Java programs

1.4.1 Writing Java code

Fortunately, writing Java code is straightforward. You can write Java code using any plain text editor. You simply need to cause the output file to have an extension of .java.

There are a number of high-level *Integrated Development Environments (IDEs)* available, such as Eclipse and NetBeans, but they tend to be overkill for the relatively simple Java programs described in these modules.

There are also some low-level IDEs available, such as JCreator and DrJava, which are very useful for sighted students. However, I don't know anything about their level of accessibility. I normally use a free version of JCreator, mainly because it contains a color-coded editor, but that feature wouldn't be useful for a blind student.

So, just find an editor that you are happy with and use it to write your Java code.

1.4.2 Preparing to compile and run Java code

Perhaps the most complicated thing is to get your computer set up for compiling and running Java code in the first place.

1.4.2.1 Downloading the java development kit *(JDK)*

You will need to download and install the free Java JDK from the Oracle/Sun website. As of August 2011, you will find that website at http://www.oracle.com/technetwork/java/javase/downloads/index.html [9]

While writing this module in August of 2011, I noticed that JDK 7 has been recently released. While I expect that it will work just fine, I haven't tried it yet. I am still running Java SE 6 Update 26. I plan to hold off for a few months before downloading and installing JDK 7.

Also there is a 64-bit version of the JDK, but I haven't tried it yet either because the computers in the labs at the college where I teach can't support it. I am still using the 32-bit version. I probably won't start using the 64-bit version until the computers in those labs are upgraded.

Whether you elect to use JDK 6 or JDK 7 in either the 32-bit or 64-bit version is strictly up to you. Any one of them should do the job very nicely.

1.4.2.2 Installing the JDK

As of August 2011, you will find installation instructions for JDK 7 at http://download.oracle.com/javase/7/docs/webnotes/install/windows/jdk-installation-windows.html [10] and you will find installation instructions for JDK 6 at http://www.oracle.com/technetwork/java/javase/index-137561.html [11] .

The installation instructions for JDK 7 are more complete than the installation instructions for JDK 6. Even if you are installing JDK 6, I recommend that you read the instructions for JDK 7 and note the additional information that you will find there, particularly the information having to do with setting the **path** environment variable.

A word of caution

If you happen to be running Windows Vista, you may need to use something like the following when updating the PATH Environment Variable

```
;C:\Program Files (x86)\Java\jdk1.6.0_26\bin
```

in place of

[9] http://www.oracle.com/technetwork/java/javase/downloads/index.html
[10] http://download.oracle.com/javase/7/docs/webnotes/install/windows/jdk-installation-windows.html
[11] http://www.oracle.com/technetwork/java/javase/index-137561.html

```
;C:\Program Files\Java\jdk1.7.0\bin
```

as shown in the installation instructions.

I don't have any experience with Windows 7 yet, so I don't have any hints regarding Windows 7. I don't have any experience with any Linux version. Therefore, I don't have any hints to offer there either.

1.4.2.3 The JDK documentation

It is very difficult to program in Java without access to the documentation for the JDK.

Several different types of Java documentation are available online at http://www.oracle.com/technetwork/java/javase/documentation/index.html [12] .

Specific documentation for classes, methods, etc., for JDK 7 is available online at http://download.oracle.com/javase/7/docs/api/ [13] . Similar documentation for JDK 6 is available at http://download.oracle.com/javase/6/docs/api/ [14] .

It is also possible to download the documentation and install it locally if you have room on your disk. The download links for JDK 6 and JDK 7 documentation are also shown on the page at http://www.oracle.com/technetwork/java/javase/downloads/index.html [15] .

1.4.3 Compiling and running Java code

There are a variety of ways to compile and run Java code. The way that I will describe here is the most basic and, in my opinion, the most reliable. These instructions apply to a Windows operating system. If you are using a different operating system, you will need to translate the instructions to your operating system.

1.4.3.1 Write your Java program

Begin by using your text editor to write your Java program into one or more text files, each with an extension of .java. *(Files of this type are often referred to as source code files.)* Save the source code files in an empty folder somewhere on your disk. Make sure that the name of the **class** containing the **main** method *(which you will learn about in a future module)* matches the name of the file in which that class is contained *(except for the extension of .java on the file name, which does not appear in the class name)* .

1.4.3.2 Create a batch file

Use your text editor to create a batch file *(or whatever the equivalent is for your operating system)* containing the text shown in Listing 1 (p. 5) *(with the modifications discussed below)* and store it in the same folder as your Java source code files..

Then execute the batch file, which in turn will execute the program if there are no compilation errors.

Listing 1.1: Windows batch file.

```
echo off
cls

del *.class

javac -cp .; hello.java
java -cp .; hello
```

[12] http://www.oracle.com/technetwork/java/javase/documentation/index.html
[13] http://download.oracle.com/javase/7/docs/api/
[14] http://download.oracle.com/javase/6/docs/api/
[15] http://www.oracle.com/technetwork/java/javase/downloads/index.html

```
pause
```

Comments regarding the batch file

The commands in the batch file of Listing 1 (p. 5) will

- Open a command-line screen for the folder containing the batch file.
- Delete all of the compiled class files from the folder. *(If the folder doesn't contain any class files, this will be indicated on the command-line screen.)*
- Attempt to compile the program in the file named **hello.java.**
- Attempt to run the compiled program using a compiled Java file named **hello.class** .
- Pause and wait for you to dismiss the command-line screen by pressing a key on the keyboard.

If errors occur, they will be reported on the command-line screen and the program won't be executed.

If your program is named something other than **hello** , *(which it typically would be)* substitute the new name for the word **hello** where it appears twice in the batch file.

Don't delete the pause command

The **pause** command causes the command-line window to stay on the screen until you dismiss it by pressing a key on the keyboard. You will need to examine the contents of the window if there are errors when you attempt to compile and run your program, so don't delete the pause command.

Translate to other operating systems

The format of the batch file in Listing 1 (p. 5) is a Windows format. If you are using a different operating system, you will need to translate the information in Listing 1 (p. 5) into the correct format for your operating system.

1.4.3.3 A test program

The test program in Listing 2 (p. 6) can be used to confirm that Java is properly installed on your computer and that you can successfully compile and execute Java programs.

Listing 1.2: A test program.

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

Instructions

Copy the code shown in Listing 2 (p. 6) into a text file named **hello.java** and store in an empty folder somewhere on your disk.

Create a batch file named **hello.bat** containing the text shown in Listing 1 (p. 5) and store that file in the same folder as the file named **hello.java** .

Execute the batch file.

If everything is working, a command-line screen should open and display the following text:

```
    Hello World
Press any key to continue . . .
```

Congratulations

If that happens, you have just written, compiled and executed your first Java program.

Oops

If that doesn't happen, you need to go back to the installation instructions and see if you can determine why the JDK isn't properly installed.

If you get an error message similar to the following, that probably means that you didn't set the **path** environment variable correctly.

```
    'javac' is not recognized as an internal or external command,
operable program or batch file.
```

Beyond that, I can't provide much advice in the way of troubleshooting hints.

1.5 Resources

I will publish a module containing consolidated links to resources on my Connexions web page and will update and add to the list as additional modules in this collection are published.

1.6 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Getting Started
- File: Jb1000.htm
- Revised: 08/18/11
- Keywords:
  - · object-oriented programming
  - · accessible
  - · accessibility
  - · blind
  - · Java
  - · screen reader
  - · refreshable Braille display

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Chapter 2

A Gentle Introduction to Java Programming[1]

2.1 Table of Contents

2.2 Preface

2.2.1 General

This module is part of a collection of modules designed to make object-oriented programming concepts accessible to blind students.

See http://cnx.org/content/col11349/latest/ [2] for the main page of the collection. If you can see the main page, there is a Table of Contents on the left side of the page. I'm not sure how your screen reader will treat that Table of Contents relative to the other parts of the page.

This module provides a gentle introduction to Java programming in a format that is accessible to blind students.

[1] This content is available online at <http://cnx.org/content/m40812/1.1/>.

[2] http://cnx.org/content/col11349/latest/

2.2.2 Prerequisites

In addition to an Internet connection and a browser, you will need the following tools *(as a minimum)* to work through the exercises in these modules:

- An audio screen reader that is compatible with your operating system, such as the NonVisual Desktop Access program (NVDA), which is freely available at http://www.nvda-project.org/ [3] .
- A refreshable Braille display capable of providing a line by line tactile output of information displayed on the computer monitor is recommended (http://www.userite.com/ecampus/lesson1/tools.php [4]).
- The Sun/Oracle Java Development Kit (JDK) *(See http://www.oracle.com/technetwork/java/javase/downloads/index.* [5] *)*
- Documentation for the Sun/Oracle Java Development Kit (JDK) *(See http://download.oracle.com/javase/7/docs/api/* [6] *)*
- A simple IDE or text editor for use in writing Java code.

The minimum prerequisites for understanding the material in these modules include:

- An understanding of algebra.
- An understanding of all of the material covered in the earlier modules in this collection.

2.2.3 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

2.2.3.1 Figures

- Figure 1 (p. 12) . A checkout counter algorithm.

2.2.3.2 Listings

- Listing 1 (p. 14) . Program named Memory01.
- Listing 2 (p. 14) . Batch file for Memory01.

2.2.4 Supplemental material

I recommend that you also study the other lessons in my extensive collection of online programming tutorials. You will find a consolidated index at www.DickBaldwin.com [7] .

2.3 Discussion and sample code

2.3.1 Introduction

All data is stored in a computer in numeric form. Computer programs do what they do by executing a series of calculations on numeric data. It is the order and the pattern of those calculations that distinguishes one computer program from another.

[3] http://www.nvda-project.org/
[4] http://www.userite.com/ecampus/lesson1/tools.php
[5] http://www.oracle.com/technetwork/java/javase/downloads/index.html
[6] http://download.oracle.com/javase/7/docs/api/
[7] http://www.dickbaldwin.com/toc.htm

Avoiding the detailed work

Fortunately, when we program using a high-level programming language such as Java, much of the detailed work is done for us behind the scenes.

Musicians or conductors

As programmers, we are more like conductors than musicians. The various parts of the computer represent the musicians. We tell them what to play, and when to play it, and if we do our job well, we produce a solution to a problem.

2.3.2 Compartments

As the computer program performs its calculations in the correct order, it is often necessary for it to store intermediate results someplace, and then come back and get them to use them in subsequent calculations later. The intermediate results are stored in memory, often referred to as RAM or *Random Access Memory* .

A mechanical analogy

We can think of random access memory as being analogous to a metal rack containing a large number of compartments. The compartments are all the same size and are arranged in a column. Each compartment has a numeric address printed above it. No two compartments have the same numeric address. Each compartment also has a little slot into which you can insert a name or a label for the compartment. No two compartments can have the same name.

Joe, the computer program

Think of yourself as a computer program. You have the ability to write values on little slips of paper and to put them into the compartments. You also have the ability to read the values written on the little slips of paper and to use those values for some purpose. However, there are two rules that you must observe:

- You may not remove a slip of paper from a compartment without replacing it by another slip of paper on which you have written a value.
- You may not put a slip of paper in a compartment without removing the one already there.

2.3.3 Checkout counter example

In understanding how you might behave as a human computer program, consider yourself to have a job working at the checkout counter of a small grocery store in the 1930s.

You have two tools to work with:

- A mechanical adding machine
- The rack of compartments described above

Initialization

Each morning, the owner of the grocery store tells you to insert a name in the slot above each compartment and to place a little slip of paper with a number written on it inside each compartment. *(In programming jargon, we would refer to this as initialization.)*

Each of the names on the compartments represents a type of grocery such as

- Beans
- Apples
- Pears

No two compartments can have the same name.

No compartment is allowed to have more than one slip of paper inside it.

The price of a can of beans

When you place a new slip of paper in a compartment, you must be careful to remove and destroy the one that was already there.

Each slip of paper that you insert into a compartment contains the price for the type of grocery identified by the label on the compartment.

For example, the slip of paper in the compartment labeled **Beans** may contain the value 15, meaning that each can of beans costs 15 cents.

The checkout procedure

As each customer comes to your checkout counter during the remainder of the day, you execute the following procedure:

- Examine each grocery item to determine its type.
- Read the price stored in the compartment corresponding to that type of grocery.
- Add that price to that customer's bill using your mechanical adding machine.

In programming jargon, we would say that as you process each grocery item for the same customer, you are looping. We would also say that you are executing a procedure or an algorithm.

When you have processed all of the grocery items for a particular customer, you would

- Press the TOTAL key on the adding machine,
- Turn the crank, and
- Present the customer with the bill.

A schematic representation of the procedure

We might represent the procedure in schematic form as shown in Figure 1 (p. 12) .

A checkout counter algorithm.

```
For each customer, do the following:

  For each item, do the following:
    a. Identify the type of grocery item
    b. Get the price from the compartment
    c. Add the price to accumulated total
  End loop on grocery items

  Present customer with a bill

End loop on a specific customer
```

Figure 2.1: A checkout counter algorithm.

Common programming activities

This procedure illustrates the three activities commonly believed to be the fundamental activities of any computer program:

- sequence
- selection
- loop

Sequence

A sequence of operations is illustrated by the three items labeled a, b, and c in Figure 1 (p. 12) because they are executed in sequential order.

Selection

The process of identifying the type of grocery item is often referred to as *selection* . A selection operation is the process of selecting among two or more choices.

Loop

The process of repetitively examining each grocery item and processing it is commonly referred to as a *loop* . In the early days of programming, for a programming language named FORTRAN, this was referred to as a ***do loop*** .

An algorithm

The entire procedure is often referred to as an *algorithm* .

Modifying stored data

Sometimes during the day, the owner of the grocery store may come to you and say that he is going to increase the price of a can of Beans from 15 cents to 25 cents and asks you to take care of the change in price.

You write 25 on a slip of paper and put it in the compartment labeled Beans, being careful to remove and destroy the slip of paper that was previously in that compartment. For the rest of the day, the new price for Beans will be used in your calculations unless the owner asks you to change it again.

Not a bad analogy

This is a pretty good analogy to how random access memory is actually used by a computer program.

Names versus addresses

As a programmer using a high-level language such as Java, you usually don't have to be concerned about the numeric addresses of the compartments.

You are able to think about them and refer to them in terms of their names. *(Names are easier to remember than numeric addresses).* However, deep inside the computer, these names are cross-referenced to addresses and at the lowest level, the program works with memory addresses instead of names.

Execute an algorithm

A computer program always executes some sort of procedure, which is often called an algorithm. The algorithm may be very simple as described in the checkout counter example described above, or it may be very complex as would be the case for a spreadsheet program. As the program executes its algorithm, it uses the random access memory to store and retrieve the data that is needed to execute the algorithm.

Why is it called RAM?

The reason this kind of memory is called *RAM* or *random access memory* is that it can be accessed in any order.

Sequential memory

Some types of memory, such as a magnetic tape, must be accessed in sequential order. This means that to get a piece of data *(the price of beans, for example)* from deep inside the memory, it is necessary to start at the beginning and examine every piece of data until the correct one is found.

Combination random/sequential

Other types of memory, such as disks, provide a combination of sequential and random access. For example, the data on a disk is stored in tracks that form concentric circles on the disk. The tracks can be accessed in random order, but the data within a track must be accessed sequentially starting at a specific point on the track.

Sequential memory isn't very good for use by most computer programs because access to each particular piece of data is quite slow.

2.3.4 Sample program

Listing 1 (p. 14) shows a sample Java program that illustrates the use of memory for the storage and retrieval of data.

Listing 2.1: Program named Memory01.

```
//File Memory01.java
class Memory01 {
  public static void main(String[] args){
    int beans;
    beans = 25;
    System.out.println(beans);
  }//end main
}//End Memory01 class
```

Listing 2 (p. 14) shows a batch file that you can use to compile and run this program.

Listing 2.2: Batch file for Memory01.

```
echo off
cls

del *.class

javac -cp .; Memory01.java
java -cp .; Memory01

pause
```

Using the procedure you learned in the module named *Getting Started* at http://cnx.org/content/m40794/latest/ [8] , you should be able to compile and execute this program. When you do, the program should display 25 on your computer screen.

Variables

You will learn in a future lesson that the term *variable* is synonymous with the term *compartment* that I have used for illustration purposes in this lesson.

The important lines of code

The use of memory is illustrated by the three lines of code in Listing 1 (p. 14) that begin with **int** , **beans** , and **System** . We will ignore the other lines in the program in this module and learn about them in future modules. *(You may find it useful to open this module in a second browser window so that you can see the program listing while you read the following discussion.)*

Declaring a variable

A memory compartment *(or variable)* is set aside and given the name **beans** by the line that begins with **int** in Listing 1 (p. 14) .

In programmer jargon, this is referred to as *declaring a variable* . The process of declaring a variable

- causes memory to be set aside to contain a value, and
- causes that chunk of memory to be given a name.

[8] http://cnx.org/content/m40794/latest/

That name can be used later to refer to the value stored in that chunk of memory or variable.

This declaration in Listing 1 (p. 14) specifies that any value stored in the variable must be of type **int** . Basically, this means that the value must be an integer. Beyond that, don't worry about what the *type* means at this point. I will explain the concept of type in detail in a future module.

Storing a value in the variable

A value of 25 is stored in the variable named **beans** by the line in Listing 1 (p. 14) that begins with the word **beans** .

In programmer jargon, this is referred to as *assigning a value to a variable* .

From this point forward, when the code in the program refers to this variable by name its name, **beans** , the reference to the variable will be interpreted to mean the value stored there.

Retrieving a value from the variable

The line in Listing 1 (p. 14) that begins with the word **System** reads the value stored in the variable named **beans** by referring to the variable by its name.

This line also causes that value to be displayed on your computer screen. However, at this point, you needn't worry about what causes it to be displayed. You will learn those details in a future module. Just remember that the reference to the variable by its name, **beans** , reads the value stored in the variable.

The remaining details

Don't be concerned at this point about the other details in the program. They are there to make it possible for you to compile and execute the program. You will learn about them in future modules.

2.4 Run the program

I encourage you to run the program that I presented in this lesson to confirm that you get the same results. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

2.5 Resources

I will publish a module containing consolidated links to resources on my Connexions web page and will update and add to the list as additional modules in this collection are published.

2.6 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: A Gentle Introduction to Java Programming
- File: Jb1010.htm
- Revised: 08/19/11
- Keywords:
  - object-oriented programming
  - accessible
  - accessibility
  - blind
  - Java
  - screen reader
  - refreshable Braille display

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Chapter 3

A Gentle Introduction to Methods in Java[1]

3.1 Table of Contents

3.2 Preface

3.2.1 General

This module is part of a collection of modules designed to make object-oriented programming concepts accessible to blind students.

See http://cnx.org/content/col11349/latest/ [2] for the main page of the collection. If you can see the main page, there is a Table of Contents on the left side of the page. I'm not sure how your screen reader will treat that Table of Contents relative to the other parts of the page.

[1] This content is available online at <http://cnx.org/content/m40817/1.1/>.
[2] http://cnx.org/content/col11349/latest/

This module provides a gentle introduction to Java programming methods in a format that is accessible to blind students.

3.2.2 Prerequisites

In addition to an Internet connection and a browser, you will need the following tools *(as a minimum)* to work through the exercises in these modules:

- An audio screen reader that is compatible with your operating system, such as the NonVisual Desktop Access program (NVDA), which is freely available at http://www.nvda-project.org/ [3] .
- A refreshable Braille display capable of providing a line by line tactile output of information displayed on the computer monitor is recommended (http://www.userite.com/ecampus/lesson1/tools.php [4]).
- The Sun/Oracle Java Development Kit (JDK) *(See http://www.oracle.com/technetwork/java/javase/downloads/index.* [5] *)*
- Documentation for the Sun/Oracle Java Development Kit (JDK) *(See http://download.oracle.com/javase/7/docs/api/* [6] *)*
- A simple IDE or text editor for use in writing Java code.

The minimum prerequisites for understanding the material in these modules include:

- An understanding of algebra.
- An understanding of all of the material covered in the earlier modules in this collection.

3.2.3 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

3.2.3.1 Listings

3.2.4 Supplemental material

I recommend that you also study the other lessons in my extensive collection of online programming tutorials. You will find a consolidated index at www.DickBaldwin.com [7] .

[3] http://www.nvda-project.org/
[4] http://www.userite.com/ecampus/lesson1/tools.php
[5] http://www.oracle.com/technetwork/java/javase/downloads/index.html
[6] http://download.oracle.com/javase/7/docs/api/
[7] http://www.dickbaldwin.com/toc.htm

3.3 Discussion and sample code

3.3.1 Introduction

Methods have been used in computer programming since the early days of programming. Methods are often called functions, procedures, subroutines, and various other names.

Calculate the square root

Suppose that your program needs to calculate the square root of a number. Referring back to your high-school algebra book, you could refresh your memory on how to calculate a square root. Then you could construct the algorithm describing that process.

Having the algorithm available, you could write the code to calculate the square root and insert it into your program code. Then you could compile, and run your program. If you did it all correctly, your program should calculate the square root. *(For reasons that will become apparent later, I will refer to the code that you inserted as in-line code.)*

Oops, need to do it all over again

Suppose that further on in your program you discover that you need to calculate the square root of another number. And later, you discover that you need to calculate the square root of still another number. Obviously, with a few changes, you could copy your original code and insert it as *in-line code* at each location in your program where you need to calculate the square root of a number.

Is there a better way?

However, after doing this a few times, you might start asking if there is a better way. The answer is *"yes, there is a better way."*

A method provides a better way

The better way is to create a separate program module that has the ability to calculate the square root and make that module available for use as a helper to your main program each time your main program needs to calculate a square root. In Java, this separate program module is called a **method** .

3.3.2 Standard methods

The Java programming language contains a large number of methods *(in the class libraries)* that are already available for your use. *(Later, I will illustrate the use of a standard method for calculating the square root of a number.)*

In addition to the standard methods that are already available, if you need a method to perform some function and there is no standard method already available to perform that function, you can write your own method.

3.3.3 Passing parameters

Make the method general

Normally, when designing and writing a method such as one that can calculate the square root of a number, it is desirable to write it in such a way that it can calculate the square root of any number *(as opposed to only one specific number)* . This is accomplished through the use of something called *parameters* .

The process of causing a method to be executed is commonly referred to as *calling the method* .

Pass me the number please

When your program calls the square-root method, it will need to tell the method the value for which the square root is needed.

In general, many methods will require that you provide certain kinds of information when you call them. The code in the method needs this information to be able to accomplish its purpose.

Passing parameters

This process of providing information to a method when you call it is commonly referred to as *passing parameters* to the method. For the square-root method, you need to pass a parameter whose value is the value of the number for which you need the square root.

3.3.4 Returning values

A method will usually

- perform an action
- send back an answer. or
- some combination of the two

Performing an action

An example of a method that performs an action is the method named **println** . We used the **println** method in an earlier module to cause information to be displayed on the computer screen. This method does not need to send back an answer, because that is not the objective of the method. The objective is simply to display some information.

Sending back an answer

On the other hand, a method that is designed to calculate the square root of a number needs to be able to send the square-root value back to the program that called the method. After all, it wouldn't be very useful if the method calculated the square root and then kept it a secret. The process of sending back an answer is commonly referred to as *returning a value* .

Returned values can be ignored

Methods can be designed in such a way that they either will or will not return a value. When a method does return a value, the program that called the method can either pay attention to that value and use it for some purpose, or ignore it entirely.

For example, in some cases where a method performs an action and also returns a value, the calling program may elect to ignore the returned value. On the other hand, if the sole purpose of a method is to return a value, it wouldn't make much sense for a program to call that method and then ignore the value that is returned *(although that would be technically possible)* .

3.3.5 Writing your own methods

As mentioned earlier, you can write your own methods in Java. I mention this here so you will know that it is possible. I will have more to say about writing your own methods in future modules.

3.3.6 Sample program

A complete listing of a sample program named **SqRt01.java** is provided in Listing 5 (p. 23) near the end of the lesson. A batch file that you can use to compile and run the program is provided in Listing 6 (p. 23) .

When you compile and run the program, the following output should appear on your computer screen:

5.049752469181039

6.0

As you will see shortly, these are the square root values respectively for 25.5 and 36.

3.3.6.1 Interesting code fragments

I will explain portions of this program in fragments. I will explain only those portions of the program that are germane to this module. Don't worry about the other details of the program. You will learn about those details in future modules.

You may find it useful to open this lesson in another browser window so that you can easily scroll back and forth among the fragments while reading the discussion.

The first code fragment that I will explain is shown in Listing 1 (p. 21) .

Listing 3.1: The price of beans.

```
double beans;
beans = 25.5;
```

What is the price of beans?

The code fragment shown in Listing 1 (p. 21) declares a variable named **beans** and assigns a value of 25.5 to the variable. *(I briefly discussed the declaration of variables in a previous module. I will discuss them in more detail in a future module.)*

What is that double thing?

In an earlier module, I declared a variable with a type named **int** . At that time, I explained that only integer values could be stored in that variable.

The variable named **beans** in Listing 1 (p. 21) is declared to be of the type **double** . I will explain the concept of data types in detail in a future module. Briefly, **double** means that you can store any numeric value in this variable, with or without a decimal part. In other words, you can store a value of 3 or a value of 3.33 in this variable, whereas a variable with a declared type of **int** won't accept a value of 3.33.

Every method has a name

Every method, every variable, and some other things as well have names. The names are case sensitive. By case sensitive, I mean that the method named **amethod** is not the same as the method named **aMethod** .

A few words about names in Java

There are several rules that define the format of allowable names in Java. You can dig into this in more detail on the web if you like, but if you follow these two rules, you will be okay:

- Use only letters and numbers in Java names.
- Always make the first character a letter.

A standard method named sqrt

Java provides a **Math** library that contains many standard methods. Included in those methods is a method named **sqrt** that will calculate and return the square root of a number that is passed as a parameter when the method is called.

The **sqrt** method is called on the right-hand side of the equal sign (=) in the code fragment in Listing 2 (p. 21) .

Listing 3.2: Compute the square root of the price of beans.

```
double sqRtBns = Math.sqrt(beans);
```

Calling the sqrt method

I'm not sure why you would want to do this, but the code fragment in Listing 2 (p. 21)

- calls the **sqrt** method and
- passes a copy of the value stored in the **beans** variable as a parameter.

The **sqrt** method calculates and returns the square root of the number that it receives as its incoming parameter. In this case, it returns the square root of the price of a can of beans.

A place to save the square root

I needed some place to save the square root value until I could display it on the computer screen later in the program. I declared another variable named **sqRtBns** in the code fragment in Listing 2 (p. 21) .

I also caused the value returned from the **sqrt** method to be stored in, or assigned to, this new variable named **sqRtBns** .

How should we interpret this code fragment?

You can think of the process implemented by the code fragment in Listing 2 (p. 21) as follows.

First note that there is an equal sign (=) near the center of the line of code. *(Later we will learn that this is called the assignment operator.)*

The code on the left-hand side of the assignment operator causes a new chunk of memory to be set aside and named **sqRtBns** . *(We call this chunk of code a variable.)*

The code on the right-hand side of the assignment operator calls the **sqrt** method, passing a copy of the value stored in the **beans** variable to the method.

When the **sqrt** method returns the value that is the square root of its incoming parameter, the assignment operator causes that value to be stored and saved in the variable named **sqRtBns** .

Now display the square root value

The code in the fragment in Listing 3 (p. 22) causes the value now stored in **sqRtBns** to be displayed on the computer screen.

Listing 3.3: Display the square root value.

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

Another method is called here

The display of the square root value is accomplished by

- calling another standard method named **println** and
- passing a copy of the value stored in **sqRtBns** as a parameter to the method.

The **println** method performs an action *(displaying something on the computer screen)* and doesn't return a value.

A method exhibits behavior

We say that a method exhibits behavior. The behavior of the **sqrt** method is to calculate and return the square root of the value passed into it as a parameter.

The behavior of the **println** method is to cause its incoming parameter to be displayed on the computer screen.

What do we mean by syntax?

Syntax is a word that is often used in computer programming. The thesaurus in the editor that I am using to type this document says that a synonym for syntax is grammar.

I also like to think of syntax as meaning something very similar to format.

Syntax for passing parameters

Note the syntax in Listing 2 (p. 21) and Listing 3 (p. 22) for passing a parameter to the method. The syntax always consists of following the name of the method with a pair of matching parentheses that contain the parameter. If more than one parameter is being passed, they are all included within the parentheses and separated by commas. Usually, the order of the parameters is important if more than one parameter is being passed.

Reusing the methods

The purpose of the code fragment in Listing 4 (p. 22) is to illustrate the reusable nature of methods.

Listing 3.4: Calling the same methods again.

```
double peas;
peas = 36.;
```

```
    double sqRtPeas = Math.sqrt(peas);
    System.out.println(sqRtPeas);
```

The code in this fragment calls the same **sqrt** method that was called before. In this case, the method is called to calculate the square root of the value stored in the variable named **peas** instead of the value stored in the variable named **beans** .

This fragment saves the value returned from the **sqrt** method in a new variable named **sqRtPeas** . Then the fragment calls the same **println** method as before to display the value now stored in the variable named **sqRtPeas** .

Write once and use over and over

Methods make it possible to write some code once and then use that code many times in the same program. This is the opposite of *in-line code* , which requires you to write essentially the same code multiple times in order to accomplish the same purpose more than once in a program.

3.4 Run the program

I encourage you to run the program that I presented in this lesson to confirm that you get the same results. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

3.5 Resources

I will publish a module containing consolidated links to resources on my Connexions web page and will update and add to the list as additional modules in this collection are published.

3.6 Complete program listings

Listing 5 (p. 23) is a complete listing of the program named **SqRt01** .

Listing 3.5: The program named SqRt01.

```
//File SqRt01.java
class SqRt01 {
  public static void main(String[] args){
    double beans;
    beans = 25.5;
    double sqRtBns = Math.sqrt(beans);
    System.out.println(sqRtBns);
    double peas;
    peas = 36.;
    double sqRtPeas = Math.sqrt(peas);
    System.out.println(sqRtPeas);
  }//end main
}//End SqRt01 class
```

Listing 6 (p. 23) contains the commands for a batch file that can be used to compile and run the program named **SqRt01** .

Listing 3.6: A batch file for compiling and running the program named SqRt01.

```
echo off
cls

del *.class

javac -cp .; SqRt01.java
java -cp .; SqRt01

pause
```

3.7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: A Gentle Introduction to Methods in Java
- File: Jb1020.htm
- Revised: 08/19/11
- Keywords:
  - object-oriented programming
  - accessible
  - accessibility
  - blind
  - Java
  - screen reader
  - refreshable Braille display
  - method

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Chapter 4

Java comments[1]

4.1 Table of Contents

4.2 Preface

4.2.1 General

This module is part of a collection of modules designed to make object-oriented programming concepts accessible to blind students.

See http://cnx.org/content/col11349/latest/ [2] for the main page of the collection. If you can see the main page, there is a Table of Contents on the left side of the page. I'm not sure how your screen reader will treat that Table of Contents relative to the other parts of the page.

This module explains Java comments in a format that is accessible to blind students.

4.2.2 Prerequisites

In addition to an Internet connection and a browser, you will need the following tools *(as a minimum)* to work through the exercises in these modules:

[1] This content is available online at <http://cnx.org/content/m40818/1.1/>.
[2] http://cnx.org/content/col11349/latest/

- An audio screen reader that is compatible with your operating system, such as the NonVisual Desktop Access program (NVDA), which is freely available at http://www.nvda-project.org/ [3] .
- A refreshable Braille display capable of providing a line by line tactile output of information displayed on the computer monitor is recommended (http://www.userite.com/ecampus/lesson1/tools.php [4]).
- The Sun/Oracle Java Development Kit (JDK) *(See http://www.oracle.com/technetwork/java/javase/downloads/index.* [5] *)*
- Documentation for the Sun/Oracle Java Development Kit (JDK) *(See http://download.oracle.com/javase/7/docs/api/* [6] *)*
- A simple IDE or text editor for use in writing Java code.

The minimum prerequisites for understanding the material in these modules include:

- An understanding of algebra.
- An understanding of all of the material covered in the earlier modules in this collection.

4.2.3 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

4.2.3.1 Figures

- Figure 1 (p. 27) . Three styles of comments.

4.2.3.2 Listings

- Listing 1 (p. 28) . A multi-line comment.
- Listing 2 (p. 28) . Three single-line comments.
- Listing 3 (p. 29) . The program named Comments01.
- Listing 4 (p. 29) . Batch file to compile and run the program named Comments01.

4.2.4 Supplemental material

I recommend that you also study the other lessons in my extensive collection of online programming tutorials. You will find a consolidated index at www.DickBaldwin.com [7] .

4.3 Discussion and sample code

4.3.1 Comments

Producing and using a Java program consists of the following steps:

1. Write the source code.
2. Compile the source code.
3. Execute the program.

[3] http://www.nvda-project.org/
[4] http://www.userite.com/ecampus/lesson1/tools.php
[5] http://www.oracle.com/technetwork/java/javase/downloads/index.html
[6] http://download.oracle.com/javase/7/docs/api/
[7] http://www.dickbaldwin.com/toc.htm

The source code consists of a set of instructions that will later be presented to a special program called a compiler for the purpose of producing a program that can be executed. In other words, when you write the source code, you are writing instructions that the compiler will use to produce the executable program.

Some things should be ignored

Sometimes, when you are writing source code, you would like to include information that may be useful to you, but should be ignored by the compiler. Information of that sort is called **comments** .

Three styles of comments

Java supports the three styles of comments shown in Figure 1 (p. 27) .

Three styles of comments.

```
/** special documentation comment
used by the javadoc tool */

/* This is a
multi-line comment */

//Single-line comment
program code // Another single-line comment
```

Figure 4.1: Three styles of comments.

The javadoc tool

The javadoc tool mentioned in Figure 1 (p. 27) is a special program that is used to produce documentation for Java program. Comments of this style begin with /** and end with */ as shown in Figure 1 (p. 27) .

The compiler ignores everything in the source code that begins and ends with this pattern of characters. Documentation produced using the javadoc program is very useful for on-line or on-screen documentation.

Multi-line comments

Multi-line comments begin with /* and end with */ as shown in Figure 1 (p. 27) . As you have probably already guessed, the compiler also ignores everything in the source code that matches this format. *(A javadoc comment is simply a multi-line comment insofar as the compiler knows. Only the special program named javadoc.exe cares about javadoc comments.)*

The multi-line comment style is particularly useful for creating large blocks of information that should be ignored by the compiler. This style can be used to produce a comment consisting of a single line of text as well. However, the single-line comment style discussed in the next section requires less typing.

Single-line comments

Single-line comments begin with // and end at the end of the line. The compiler ignores the // and everything following the slash characters to the end of the line.

This style is particularly useful for inserting short comments throughout the source code. In this case, the // can appear at the beginning of the line as shown in Figure 1 (p. 27) , or can appear anywhere in the line, including at the end of some valid source code *(also shown in Figure 1* (p. 27) *)* .

4.3.2 Sample program

The purpose of the program named **Comments01** , which is shown in Listing 3 (p. 29) near the end of the module, is to illustrate the use of single and multi-line comments. The program does not contain any javadoc comments.

The commands for a batch file that you can use to compile and run this program are provided in Listing 4 (p. 29) .

When you compile and run the program, the following text should appear on your command-line screen:

Hello World

4.3.2.1 Interesting code fragments

I will explain this program in fragments, and will explain only those portions of the program that are germane to this module. Don't worry about the other details of the program at this time. You will learn about those details in future modules.

A multi-line comment

Listing 1 (p. 28) , shows a multi-line comment, which consists of three lines of text.

As required, this multi-line comment begins with /* and ends with */. The extra stars on the third line are simply part of the comment.

You will often see formats similar to this being used to provide a visual separation between multi-line comments and the other parts of a program.

Listing 4.1: A multi-line comment.

```
/*File Comments01.java
This is a multi-line comment.
*****************************************/
```

Single-line comments

Listing 2 (p. 28) shows three single-line comments. Can you spot them? Remember, single-line comments begin with //.

Listing 4.2: Three single-line comments.

```
class Comments01 {
  //This is a single-line comment
  public static void main(String[] args){
    System.out.println("Hello World");
  }//end main
}//End class
```

One of the comments in Listing 2 (p. 28) starts at the beginning of the line. The other two comments follow some program code.

4.4 Run the program

I encourage you to run the program that I presented in this lesson to confirm that you get the same results. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

4.5 Resources

I will publish a module containing consolidated links to resources on my Connexions web page and will update and add to the list as additional modules in this collection are published.

4.6 Complete program listings

Listing 3 (p. 29) contains a complete listing of the program named **Comments01** .

Listing 4.3: The program named Comments01.

```
/*File Comments01.java
This is a multi-line comment.
****************************************/
class Comments01 {
  //This is a single-line comment
  public static void main(String[] args){
    System.out.println("Hello World");
  }//end main
}//End class
```

Listing 4 (p. 29) contains the commands for a batch file that can be used to compile and run the program named **Comments01** .

Listing 4.4: Batch file to compile and run the program named Comments01.

```
echo off
cls

del *.class

javac -cp .; Comments01.java
java -cp .; Comments01

pause
```

4.7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java Comments
- File: Jb1030.htm
- Revised: 08/19/11
- Keywords:
  - object-oriented programming
  - accessible

- accessibility
- blind
- Java
- screen reader
- refreshable Braille display
- OOP
- comment

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Chapter 5

Java Data Types[1]

5.1 Table of Contents

5.2 Preface

5.2.1 General

This module is part of a collection of modules designed to make object-oriented programming concepts accessible to blind students.

See http://cnx.org/content/col11349/latest/ [2] for the main page of the collection. If you can see the main page, there is a Table of Contents on the left side of the page. I'm not sure how your screen reader will treat that Table of Contents relative to the other parts of the page.

This module explains Java data types in a format that is accessible to blind students.

[1] This content is available online at <http://cnx.org/content/m40872/1.2/>.
[2] http://cnx.org/content/col11349/latest/

5.2.2 Prerequisites

In addition to an Internet connection and a browser, you will need the following tools *(as a minimum)* to work through the exercises in these modules:

- An audio screen reader that is compatible with your operating system, such as the NonVisual Desktop Access program (NVDA), which is freely available at http://www.nvda-project.org/ [3] .
- A refreshable Braille display capable of providing a line by line tactile output of information displayed on the computer monitor is recommended (http://www.userite.com/ecampus/lesson1/tools.php [4]).
- The Sun/Oracle Java Development Kit (JDK) *(See http://www.oracle.com/technetwork/java/javase/downloads/index.* [5] *)*
- Documentation for the Sun/Oracle Java Development Kit (JDK) *(See http://download.oracle.com/javase/7/docs/api/* [6] *)*
- A simple IDE or text editor for use in writing Java code.

The minimum prerequisites for understanding the material in these modules include:

- An understanding of algebra.
- An understanding of all of the material covered in the earlier modules in this collection.

5.2.3 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures while you are reading about them.

5.2.3.1 Figures

- Figure 1 (p. 36) . Range of values for whole-number types.
- Figure 2 (p. ??) . Definition of floating point.
- Figure 3 (p. 37) . Different ways to represent 623.57185.
- Figure 4 (p. 38) . Relationships between multiplicative factors and exponentiation.
- Figure 5 (p. 39) . Other ways to represent the same information.
- Figure 6 (p. 39) . Still other ways to represent 623.57185.
- Figure 7 (p. 41) . Range of values for floating-point types.
- Figure 8 (p. 42) . Example of the use of the boolean type.

5.2.4 Supplemental material

I recommend that you also study the other lessons in my extensive collection of online programming tutorials. You will find a consolidated index at www.DickBaldwin.com [7] .

5.3 Discussion

5.3.1 Introduction

Type-sensitive languages

[3] http://www.nvda-project.org/
[4] http://www.userite.com/ecampus/lesson1/tools.php
[5] http://www.oracle.com/technetwork/java/javase/downloads/index.html
[6] http://download.oracle.com/javase/7/docs/api/
[7] http://www.dickbaldwin.com/toc.htm

Java and some other modern programming languages make heavy use of a concept that we refer to as *type* , or *data type* .

We refer to those languages as *type-sensitive languages* . Not all languages are type-sensitive languages. In particular, some languages hide the concept of type from the programmer and automatically deal with type issues behind the scenes.

So, what do we mean by type?

One analogy that comes to my mind is international currency. For example, many years ago, I spent a little time in Japan and quite a long time on an island named Okinawa *(Okinawa is now part of Japan)* .

Types of currency

At that time, as now, the type of currency used in the United States was the dollar. The type of currency used in Japan was the yen, and the type of currency used on the island of Okinawa was also the yen. However, even though two of the currencies had the same name, they were different types of currency, as determined by the value relationships among them.

The exchange rate

As I recall, at that time, the exchange rate between the Japanese yen and the U.S. dollar was 360 for each dollar. The exchange rate between the Okinawan yen and the U.S. dollar was 120 yen for each dollar. This suggests that the exchange rate between the Japanese yen and the Okinawan yen would have been 3 Japanese yen for each Okinawan yen.

Analogous to different types of data

So, why am I telling you this? I am telling you this to illustrate the concept that different types of currency are roughly analogous to different data types in programming.

Purchasing transactions were type sensitive

In particular, because there were three different types of currency involved, the differences in the types had to be taken into account in any purchasing transaction to determine the price in that particular currency. In other words, the purchasing process was sensitive to the type of currency being used for the purchase *(type sensitive)* .

Different types of data

Type-sensitive programming languages deal with different types of data. Some data types such at type **int** involve whole numbers only *(no fractional parts are allowed)* .

Other data types such as **double** involve numbers with fractional parts.

Some data types conceptually have nothing to do with numeric values, but deal only with the concept of true or false *(**boolean**)* or with the concept of the letters of the alphabet and the punctuation characters *(**char**)* .

Type specification

For every different type of data used with a particular programming language, there is a specification somewhere that defines two important characteristics of the type:

1. What is the set of all possible data values that can be stored in an instance of the type *(we will learn some other names for instance later)* ?
2. Once you have an instance of the type, what are the operations that you can perform on that instance alone, or in combination with other instances?

What do I mean by an instance of a type?

Think of the type specification as being analogous to the plan or blueprint for a model airplane. Assume that you build three model airplanes from the same set of plans. You will have created three instances of the plans.

We might say that an instance is the physical manifestation of a plan or a type.

Using mixed types

Somewhat secondary to the specifications for the different types, but also extremely important, is a set of rules that define what happens when you perform an operation involving mixed types *(such as making a purchase using some yen currency in combination with some dollar currency)* .

The short data type

For example, in addition to the integer type **int** , there is a data type in Java known as **short** . The **short** type is also an integer type.

If you have an instance of the **short** type, the set of all possible values that you can store in that instance is the set of all the whole numbers ranging from -32,768 to +32,767.

This constitutes a set of 65,536 different values, including the value zero. No other value can be stored in an instance of the type **short** . For example, you cannot store the value 35,000 in an instance of the type **short** in Java. If you need to store that value, you will need to use some type other than **short** .

Kind of like an odometer

This is somewhat analogous to the odometer in your car *(the thing that records how many miles the car has been driven)* . For example, depending on the make and model of car, there is a specified set of values that can appear in the odometer. The value that appears in the odometer depends on how many miles your car has been driven.

It is fairly common for an odometer to be able to store and to display the set of all positive values ranging from zero to 99999. If your odometer is designed to store that set of values and if you drive your car more than 99999 miles, it is likely that the odometer will roll over and start back at zero after you pass the 99999-mile mark. In other words, that particular odometer does not have the ability to store a value of 100,000 miles. Once you pass the 99999-mark, the data stored in the odometer is corrupt.

Now let's return to the Java type named short

Assume that you have two instances of the type **short** in a Java program. What are the operations that you can perform on those instances? For example:

- You can add them together.
- You can subtract one from the other.
- You can multiply one by the other.
- You can divide one by the other.
- You can compare one with the other to determine which is algebraically larger.

There are some other operations that are allowed as well. In fact, there is a well-defined set of operations that you are allowed to perform on those instances. That set of operations is defined in the specification for the type **short** .

What if you want to do something different?

However, if you want to perform an operation that is not allowed by the type specification, then you will have to find another way to accomplish that purpose.

For example, some programming languages allow you to raise whole-number types to a power *(examples: four squared, six cubed, nine to the fourth power, etc.)* . However, that operation is not allowed by the Java specification for the type **short** . If you need to do that operation with a data value of the Java **short** type, you must find another way to do it.

Two major categories of type

Java data types can be subdivided into two major categories:

- Primitive types
- User-defined or reference types

These categories are discussed in more detail in the following sections.

5.3.2 Primitive types

Java is an extensible programming language

What this means is that there is a core component to the language that is always available. Beyond this, individual programmers can extend the language to provide new capabilities. The primitive types discussed in this section are the types that are part of the core language. A later section will discuss user-defined types that become available when a programmer extends the language.

More subdivision

It seems that when teaching programming, I constantly find myself subdividing topics into sub-topics. I am going to subdivide the topic of Primitive Types into four categories:

- Whole-number types
- Floating-point types
- Character types
- Boolean types

Hopefully this categorization will make it possible for me to explain these types in a way that is easier for you to understand.

5.3.2.1 Whole-number types

The whole-number types, often called *integer* types, are relatively easy to understand. These are types that can be used to represent data without fractional parts.

Applesauce and hamburger

For example, consider purchasing applesauce and hamburger. At the grocery store where I shop, I am allowed to purchase cans of applesauce only in whole-number or integer quantities.

Can purchase integer quantities only

For example, the grocer is happy to sell me one can of applesauce and is even happier to sell me 36 cans of applesauce. However, she would be very unhappy if I were to open a can of applesauce in the store and attempt to purchase 6.3 cans of applesauce.

Counting doesn't require fractional parts

A count of the number of cans of applesauce that I purchase is somewhat analogous to the concept of whole-number data types in Java. Applesauce is not available in fractional parts of cans *(at my grocery store)* .

Fractional pounds of hamburger are available

On the other hand, the grocer is perfectly willing to sell me 6.3 pounds of hamburger. This is somewhat analogous to *floating-point data types* in Java.

Accommodating applesauce and hamburger in a program

Therefore, if I were writing a program dealing with quantities of applesauce and hamburger, I might elect to use a whole number type to represent cans of applesauce and to use a floating-point type to represent pounds of hamburger.

Different whole-number types

In Java, there are four different whole-number types:

- byte
- short
- int
- long

(The char type is also a whole number type, but since it is not intended to be used for arithmetic, I discuss it later as a character type.)

The four types differ primarily in terms of the range of values that they can accommodate and the amount of computer memory required to store instances of the types.

Differences in operations?

Although there are some subtle differences among the four whole-number types in terms of the operations that you can perform on them, I will defer a discussion of those differences until a more advanced module. *(For example some operations require instances of the* ***byte*** *and* ***short*** *types to be converted to type* ***int*** *before the operation takes place.)*

Algebraically signed values

All four of these types can be used to represent algebraically signed values ranging from a specific negative value to a specific positive value.

Range of the byte type

For example, the **byte** type can be used to represent the set of whole numbers ranging from -128 to +127 inclusive. *(This constitutes a set of 256 different values, including the value zero.)*

The **byte** type cannot be used to represent any value outside this range. For example, the **byte** type cannot be used to represent either -129 or +128.

No fractional parts allowed by the byte type

Also, the **byte** type cannot be used to represent fractional values within the allowable range. For example, the byte type cannot be used to represent the value of 63.5 or any other value that has a fractional part.

Like a strange odometer

To form a crude analogy, the byte type is sort of like a strange odometer in a new *(and unusual)* car that shows a mileage value of -128 when you first purchase the car. As you drive the car, the negative values shown on the odometer increment toward zero and then pass zero. Beyond that point they increment up toward the value of +127.

Oops, numeric overflow!

When the value passes *(or attempts to pass)* +127 miles, something bad happens. From that point forward, the value shown on the odometer is not a reliable indicator of the number of miles that the car has been driven.

Ranges for each of the whole-number types

Figure 1 (p. 36) shows the range of values that can be accommodated by each of the four whole-number types supported by the Java programming language:

Range of values for whole-number types.

```
byte
-128 to +127

short
-32768 to +32767

int
-2147483648 to +2147483647

long
-9223372036854775808 to +9223372036854775807
```

Figure 5.1: Range of values for whole-number types.

Can represent some fairly large values

As you can see, the **int** and **long** types can represent some fairly large values. However, if your task involves calculations such as distances in interstellar space *(or the U.S. national debt)* , these ranges probably won't accommodate your needs. This will lead you to consider using the *floating-point* types

discussed in the upcoming sections. I will discuss the operations that can be performed on whole-number types more fully in future modules.

5.3.2.2 Floating-point types

Floating-point types are a little more complicated than whole-number types. I found the definition of floating-point shown in Figure 2 (p. ??) in the *Free On-Line Dictionary of Computing* at this URL [8] .

A number representation consisting of a mantissa, M, an exponent, E, and an (assumed) radix *(or "base")* . The numbe

Figure 5.2: Definition of floating point.

So what does this really mean?
Assuming a base or radix of 10, I will attempt to explain it using an example.
Consider the following value:
623.57185
I can represent this value in any of the ways shown in Figure 3 (p. 37) *(where * indicates multiplication)*.

Different ways to represent 623.57185.

```
.62357185*1000
6.2357185*100
62.357185*10
623.57185*1
6235.7185*0.1
62357.185*0.01
623571.85*0.001
6235718.5*0.0001
62357185.*0.00001
```

Figure 5.3: Different ways to represent 623.57185.

[8] http://foldoc.org/floating+point

In other words, I can represent the value as a mantissa (62357185) multiplied by a factor where the purpose of the factor is to represent a left or right shift in the position of the decimal point.

Now consider the factor

Each of the factors shown in Figure 3 (p. 37) represents the value of ten raised to some specific power, such as ten squared, ten cubed, ten raised to the fourth power, etc.

Exponentiation

If we allow the following symbol (^) to represent exponentiation *(raising to a power)* and allow the following symbol (/) to represent division, then we can write the values for the above factors in the ways shown in Figure 4 (p. 38) .

Note in particular the characters following the first equal character (=) on each line, which I will refer to later as the exponents.

Relationships between multiplicative factors and exponentiation.

```
    1000 = 10^+3 = 1*10*10*10
100 = 10^+2 = 1*10*10
10 = 10^+1 = 1*10
1 = 10^+0 = 1
0.1 = 10^-1 = 1/10
0.01 = 10^-2 = 1/(10*10)
0.001 = 10^-3 = 1/(10*10*10)
0.0001 = 10^-4 = 1/(10*10*10*10)
0.00001 = 10^-5 = 1/(10*10*10*10*10)
```

Figure 5.4: Relationships between multiplicative factors and exponentiation.

In the above notation, the term 10^+3 means 10 raised to the third power.

The zeroth power

By definition, the value of any value raised to the zeroth power is 1. *(Check this out in your high-school algebra book.)*

The exponent and the factor

Hopefully, at this point you will understand the relationship between the exponent and the factor introduced earlier in Figure 3 (p. 37) .

Different ways to represent the same value

Having reached this point, by using substitution, I can rewrite the original set of representations (p. 37) of the value 623.57185 in the ways shown in Figure 5 (p. 39) . It is very important to for you to understand that these are simply different ways to represent the same value.

Other ways to represent the same information.

```
    .62357185*10^+3
6.2357185*10^+2
62.357185*10^+1
623.57185*10^+0
6235.7185*10^-1
62357.185*10^-2
623571.85*10^-3
6235718.5*10^-4
62357185.*10^-5
```

Figure 5.5: Other ways to represent the same information.

A simple change in notation

Finally, by making a simplifying change in notation where I replace (*10^) by (E) I can rewrite the different representations of the value of 623.57185 in the ways shown in Figure 6 (p. 39) .

Still other ways to represent 623.57185.

```
    .62357185E+3
6.2357185E+2
62.357185E+1
623.57185E+0
6235.7185E-1
62357.185E-2
623571.85E-3
6235718.5E-4
62357185.E-5
```

Figure 5.6: Still other ways to represent 623.57185.

Getting the true value

Floating point types represent values as a mantissa containing a decimal point along with an exponent

value which tells how many places to shift the decimal point to the left or to the right in order to determine the true value.

Positive exponent values mean that the decimal point should be shifted to the right. Negative exponent values mean that the decimal point should be shifted to the left.

Maintaining fractional parts

One advantage of floating-point types is that they can be used to maintain fractional parts in data values, such as 6.3 pounds of hamburger.

Accommodating a very large range of values

Another advantage is that a very large range of values can be represented using a reasonably small amount of computer memory for storage of the values.

Another example

For example *(assuming that I counted the number of digits correctly)* the following very large value
62357185000000000000000000000000000000.0
can be represented as
6.2357185E+37
Similarly, again assuming that I counted the digits correctly, the following very small value
0.00000000000000000000000000000062357185
can be represented as
6.2357185E-30

When would you use floating-point?

If you happen to be working in an area where you

- need to keep track of fractional parts *(such as the amount of hamburger in a package)* ,
- have to work with extremely large numbers *(distances between galaxies)* , or
- have to work with extremely small values *(the size of atomic particles)* ,

then you will need to use the floating-point types.

Don't use floating-point in financial transactions

You probably don't want to use floating-point in financial calculations, however, because there is a lot of rounding that takes place in floating-point calculations. In other words, floating point calculations provide answers that are very close to the truth but the answers are often not exact.

Two floating-point types

Java supports two different floating point types:

- float
- double

These two types differ primarily in terms of the range of values that they can support.

Range of values for floating point types

The table in **Figure 7** (p. 41) shows the smallest and largest values that can be accommodated by each of the floating-point types. Values of either type can be either positive or negative.

Range of values for floating-point types.

```
    float
1.4E-45 to 3.4028235E38

double
4.9E-324 to 1.7976931348623157E308
```

Figure 5.7: Range of values for floating-point types.

I will discuss the operations that can be performed on floating-point types in a future module.

5.3.2.3 The character type

Computers deal only in numeric values. They don't know how to deal directly with the letters of the alphabet and punctuation characters. This gives rise to a type named **char** .

Purpose of the char type

The purpose of the character type is to make it possible to represent the letters of the alphabet, the punctuation characters, and the numeric characters internally in the computer. This is accomplished by assigning a numeric value to each character, much as you may have done to create secret codes when you were a child.

A single character type

Java supports a single character type named **char** . The char type uses a standard character representation known as **Unicode** to represent up to 65,535 different characters.

Why so many characters?

The reason for the large number of possible characters is to make it possible to represent the characters making up the alphabets of many different countries and many different languages.

What are the numeric values representing characters?

As long as the characters that you use in your program appear on your keyboard, you usually don't have a need to know the numeric value associated with the different characters. If you are curious, however, the upper-case A is represented by the value 65 in the Unicode character set.

Representing a character symbolically

In Java, you usually represent a character in your program by surrounding it with apostrophes as shown below:

'A'.

The Java programming tools know how to cross reference that specific character symbol against the Unicode table to obtain the corresponding numeric value. *(A discussion of the use of the* ***char*** *type to represent characters that don't appear on your keyboard is beyond the scope of this module.)*

I will discuss the operations that can be performed on the **char** type in a future module.

5.3.2.4 The boolean type

The boolean type is the simplest type supported by Java. It can have only two values:

- true
- false

Generally speaking, about the only operations that can be directly applied to an instance of the **boolean** type are to change it from **true** to **false** , and vice versa. However, the **boolean** type can be included in a large number of somewhat higher-level operations.

The **boolean** type is commonly used in some sort of a test to determine what to do next, such as that shown in Figure 8 (p. 42) .

Example of the use of the boolean type.

```
Perform a test that returns a value of type boolean.
if that value is true,
  do one thing
otherwise (meaning that value is false)
  do a different thing
```

Figure 5.8: Example of the use of the boolean type.

I will discuss the operations that can be performed on the boolean type in more detail in a future module.

5.3.3 User-defined or reference types

Extending the language

Java is an *extensible* programming language. By this, I mean that there is a core component to the language that is always available. Beyond the core component, different programmers can extend the language in different ways to meet their individual needs.

Creating new types

One of the ways that individual programmers can extend the language is to create new types. When creating a new type, the programmer must define the set of values that can be stored in an instance of the type as well as the operations that can be performed on instances of the type.

No magic involved

While this might initially seem like magic, once you get to the heart of the matter, it is really pretty straight forward. New types are created by combining instances of primitive types along with instances of other user-defined types. In other words, the process begins with the primitive types explained earlier and builds upward from there.

An example

For example, a **String** type, which can be used to represent a person's last name, is just a grouping of a bunch of instances of the primitive **char** or character type.

A user-defined **Person** type, which could be used to represent both a person's first name and their last name, might simply be a grouping of two instances of the user-defined **String** type.

Differences

The biggest conceptual difference between the **String** type and the **Person** type is that the **String** type is contained in the standard Java library while the **Person** type isn't in that library. However, you could put it in a library of your own design if you choose to do so.

Removing types

You could easily remove the **String** type from your copy of the standard Java library if you choose to do so, although that would probably be a bad idea. However, you cannot remove the primitive **double** type from the core language without making major modifications to the language.

The company telephone book

A programmer responsible for producing the company telephone book might create a new type that can be used to store the first and last names along with the telephone number of an individual. That programmer might choose to give the new type the name **Employee** .

The programmer could create an instance of the **Employee** type to represent each employee in the company, populating each such instance with the name and telephone number for an individual employee. *(At this point, let me sneak a little jargon in and tell you that we will be referring to such instances as objects.)*

A comparison operation

The programmer might define one of the allowable operations for the new **Employee** type to be a comparison between two objects of the new type to determine which is greater in an alphabetical sorting sense. This operation could be used to sort the set of objects representing all of the employees into alphabetical order. The set of sorted objects could then be used to print a new telephone book.

A name-change operation

Another allowable operation that the programmer might define would be the ability to change the name stored in an object representing a particular employee. For example when Suzie Smith marries Tom Jones, she might elect to thereafter be known as

- Suzie Jones,
- Suzie Smith-Jones,
- Suzie Jones-Smith, or
- something entirely different.

In this case, there would be a need to modify the object that represents Suzie in order to reflect her newly-elected surname. *(Or perhaps Tom Jones might elect to thereafter be known as Tom Smith, in which case it would be necessary to modify the object that represents him.)*

An updated telephone book

The person charged with maintaining the database could

- use the name-changing operation to modify the object and change the name,
- make use of the sorting operation to re-sort the set of objects, and
- print and distribute an updated version of the telephone book.

Many user-defined types already exist

Unlike the primitive types which are predefined in the core language, I am unable to give you much in the way of specific information about user-defined types, simply because they don't exist until a user defines them.

I can tell you, however, that when you obtain the Java programming tools from Sun, you not only receive the core language containing the primitive types, you also receive a large library containing several thousand user-defined types that have already been defined. A large documentation package is available from Sun to help you determine the individual characteristics of these user-defined types.

The most important thing

At this stage in your development as a Java programmer, the most important thing for you to know about user-defined types is that they are possible.

You can define new types. Unlike earlier procedural programming languages such as C and Pascal, you are no longer forced to adapt your problem to the available tools. Rather, you now have the opportunity to extend the tools to make them better suited to solve your problem.

The class definition

The specific Java mechanism that makes it possible for you to define new types is a mechanism known as the *class definition* .

In Java, whenever you define a new class, you are at the same time defining a new type. Your new type can be as simple, or as complex and powerful as you want it to be.

An object *(instance)* of your new type can contain a very small amount of data, or it can contain a very large amount of data. The operations that you allow to be performed on an object of your new type can be rudimentary, or they can be very powerful.

It is all up to you

Whenever you define a new class *(type)* you not only have the opportunity to define the data definition and the operations, you also have a responsibility to do so.

Much to learn and much to do

But, you still have much to learn and much to do before you will need to define new types.

There are a lot of fundamental programming concepts that we will need to cover before we seriously embark on a study involving the definition of new types.

For the present then, simply remember that such a capability is available, and if you work to expand your knowledge of Java programming one small step at a time, when we reach the point of defining new types, you will be ready and eager to do so.

5.3.4 Sample program

I'm not going to provide a sample program in this module. Instead, I will be using what you have learned about Java data types in the sample programs in future modules.

5.4 Resources

I will publish a module containing consolidated links to resources on my Connexions web page and will update and add to the list as additional modules in this collection are published.

5.5 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java Data Types
- File: Jb1040.htm
- Revised: 08/21/11
- Keywords:
  - object-oriented programming
  - accessible
  - accessibility
  - blind
  - Java
  - screen reader
  - refreshable Braille display
  - data types

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Index of Keywords and Terms

Keywords are listed by the section with that keyword (page numbers are in parentheses). Keywords do not necessarily appear in the text of the page. They are merely associated with that section. *Ex.* apples, § 1.1 (1) **Terms** are referenced by the page they appear on. *Ex.* apples, 1

Attributions

Collection: *Accessible Objected-Oriented Programming Concepts for Blind Students*
Edited by: Richard Baldwin
URL: http://cnx.org/content/col11349/1.6/
License: http://creativecommons.org/licenses/by/3.0/

Module: "Getting Started"
By: Richard Baldwin
URL: http://cnx.org/content/m40794/1.1/
Pages: 1-7
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "A Gentle Introduction to Java Programming"
By: Richard Baldwin
URL: http://cnx.org/content/m40812/1.1/
Pages: 9-16
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "A Gentle Introduction to Methods in Java"
By: Richard Baldwin
URL: http://cnx.org/content/m40817/1.1/
Pages: 17-24
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java comments"
By: Richard Baldwin
URL: http://cnx.org/content/m40818/1.1/
Pages: 25-30
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Module: "Java Data Types"
By: Richard Baldwin
URL: http://cnx.org/content/m40872/1.2/
Pages: 31-45
Copyright: Richard Baldwin
License: http://creativecommons.org/licenses/by/3.0/

Accessible Objected-Oriented Programming Concepts for Blind Students
Blind students should not be excluded from computer programming courses because of inaccessible textbooks. Because of its text-based nature, computer programming is fundamentally an accessible technology. However, many textbooks adopt and use high-level integrated development environments with graphical user interfaces that greatly reduce that accessibility. The modules in this collection present object-oriented programming concepts in a format that blind students can read using tools such as an audio screen reader and an electronic line-by-line Braille display. In an effort to get and keep the student's interest, these modules make heavy use of programming projects that provide sensory feedback through the use of both sampled sound and MIDI sound. These modules are intended to supplement and not to replace the textbook.