

The Java Learning Kit: Chapter 2 Reading, Writing, and Arithmetic

Lesson 2.1 – Data Types

Lesson 2.2 – Variables and Constants

Lesson 2.3 – Assignment Statements and Expressions

Lesson 2.4 – Data Streams and System Console I/O

Lesson 2.5 – Documentation First Programming

Lab 2A – Console I/O: Age in Days

Lesson 2.6 – JOptionPane Pop-Up Dialog Windows

Lab 2B – Pop Up Windows Dialog: Age in Days

The Java Learning Kit: Chapter 2 – Reading, Writing, and Arithmetic

Copyright 2015 by C. Herbert, all rights reserved.

Last edited January, 2015 by C. Herbert

This document is a chapter from a draft of the Java Learning Kit, written by Charles Herbert, with editorial input from Craig Nelson, Christopher Quinones, Matthew Staley, and Daphne Herbert. It is available free of charge for students in Computer Science courses at Community College of Philadelphia during the Spring 2015 semester.

This material is protected by United States and international copyright law and may not be reproduced, distributed, transmitted, displayed, published or broadcast without the prior written permission of the copyright holder. You may not alter or remove any trademark, copyright or other notice from copies of the material.

Contents

Chapter 2 Learning Outcomes	4
Lesson 2.1 Data Types.....	5
Common Data Types in Java	5
Variables, Constants, and Literals	6
Boolean Data	6
Text Data.....	7
Special Characters in Strings	9
Integer Data	10
Floating Point Data	10
IEEE Floating Point Numbers.....	11
<i>CheckPoint 2.1</i>	12
Numbers from the Real (and Imaginary) World.....	13
Lesson 2.2 Variables and Constants	14
Local Variables	14
Constant Declarations.....	15
Hardcoding.....	15
Variable and Constant Names – Java Identifiers	15
<i>CheckPoint 2.2</i>	16
Lesson 2.3 Assignment Statements and Expressions	17
Polymorphism.....	17
Arithmetic Operations	18
Quotients and Remainders	19
Unary Operations.....	19
The Java Math Class.....	20
Order of Operations.....	22
Commutative and Associative Behavior.....	24
Type Casting.....	25
Cast Operators	26
CheckPoint 2.3	26
Lesson 2.4 Data Streams and System Console I/O	27
The Scanner Class	28
Checkpoint 2.4	29
Lesson 2.5 Documentation First Programming	30

Lab 2A – Console I/O: Age in Days	33
Lesson 2.6 JOptionPane Pop-Up Dialog Windows.....	38
Using the JOptionPane Message Dialog Window.....	38
Using the JOptionPane Input Dialog Window	40
Parsing Numbers from Strings	41
Exceptions When Parsing Numbers from Strings	43
Lab 2B – Pop Up Windows Dialog: Age in Days	44
Key Terms in Chapter 2	48
Chapter 2 - Questions	49
Chapter 2 – Exercises	50

The Java Learning Kit: Chapter 2

Reading, Writing, and Arithmetic

This chapter introduces some of the basic elements needed to create Java programs. Lessons 1, 2 and 3 discuss data types and simple arithmetic in Java, including assignment statements and math expressions.

Lesson 4 looks at console I/O, which is text input and output based on data streams. We won't learn how to read and write data files in this chapter - that comes later - but the text data streams used for reading input from the keyboard and writing output to the screen are the basis for working with text data files in Chapter 6.

Lesson 5 shows an alternative method of I/O - using pop-up dialog windows, which are more interesting and attractive than console I/O, and are fairly simple to use.

Lesson 6 is about the importance of documentation and how to use documentation as tool to help design and build software.

The two labs in the chapter will provide you with some practice using most of the material covered in the chapter. The first lab is based on console I/O and the second lab uses pop-up dialog windows.

Chapter 2 Learning Outcomes

Upon completion of this chapter students should be able to:

- describe the concept of a data type; list and describe the primitive data types in Java;
- describe the difference between primitive data types and classes of objects, and how the String data type is unique in this regard;
- describe the mathematical concepts of integers, rational numbers, and real numbers, and how they are implemented in Java;
- describe the concepts of variables and constants, scope rules for variables, and Java naming conventions for variables;
- declare and assign values to primitive and String variables in Java;
- describe the benefits of frontloading declarations in Java methods;
- describe the concept of type casting and how type casting works in Java;
- create Java expressions to perform integer and floating point addition, subtraction, multiplication and division, and the remainder operation for integers;
- create Java code that uses methods from the `java.lang.Math` class;
- describe the concept of a data stream and stream tokenization, and how console input and output data streams are handled in Java;
- create Java code to allow applications to read data from a keyboard as console input;
- describe what a pop-up dialog window is and how to create message and input dialogs windows in Java, including how to parse numbers from input Strings.
- create Java code that implements a dialog with the user via `JOptionPane` pop-up dialog windows.

Lesson 2.1 Data Types

A **data type** is a format used to store data, including instructions for operations that can be performed on the data, along with an acceptable range of values for the data.

Formats for data depend on what the data means and how the data will be used. Numbers, for example, are stored in formats that allow the computer to quickly perform arithmetic with the data. Using different data types allows computers to process data more efficiently – using less memory to store data and less time to process data.

Generally, object-oriented programming languages have two main categories of data types – *primitive data types* and *reference data types*. A **primitive data type** is built into the programming language and may be used by a programmer without first defining the data type. A **reference data type** refers to an object, which needs to be defined by a programmer before it can be used. Reference data types are also called user-defined data types. Each class of object in Java is essentially its own user-defined data type. There is one exception – the *String* class in Java is a special class of objects, built into the language. We may use *String* as a primitive data type, even though it is actually a reference data type. The primitive data types and the *String* class are together known as Java’s **built-in data types**.

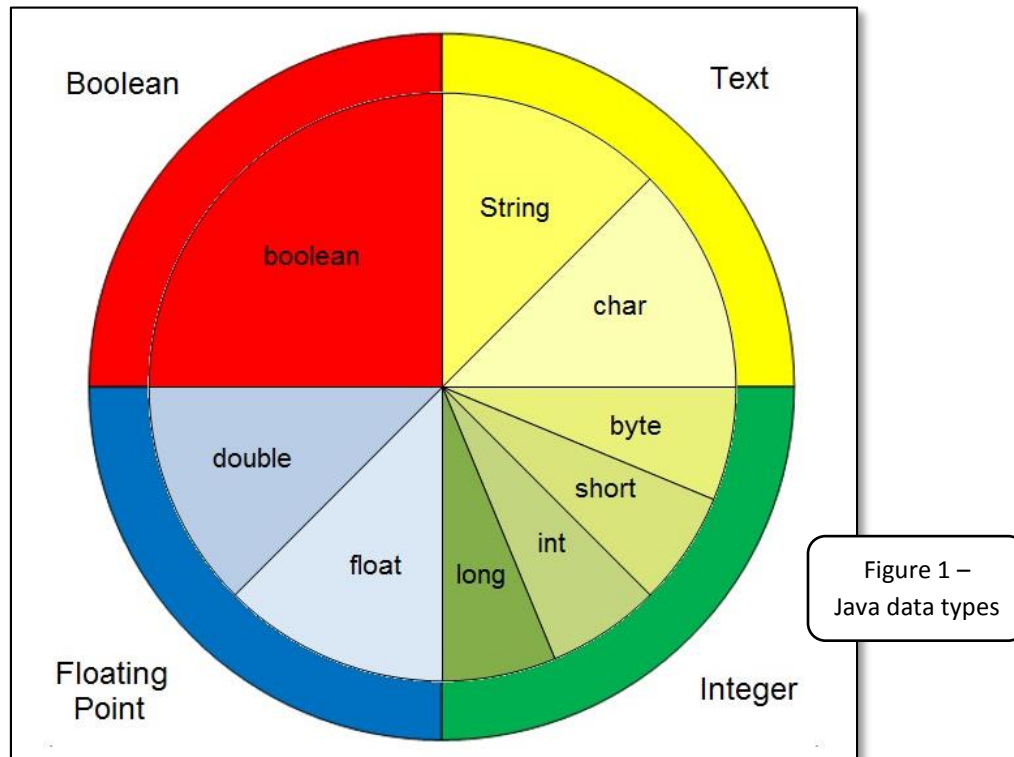
Common Data Types in Java

Most programming languages have data types that each fall into one of four different categories of data:

- *Boolean data* is used to keep track of true and false values.
- *text data* is composed of characters – the symbols we use in human languages, such as the Latin alphabet used for the English language.
- *integer data* is numeric data without fractions.
- floating point data is numeric with fractions, most often as decimal numbers with a decimal point.

Java has nine built-in data types that each fall into one of these categories, as shown Figure 1. Eight of built-in data types shown in the chart – boolean, char, byte, short, int, long, float, and double – are Java’s primitive data types, described in this chapter. The ninth built-in datatype is the *String* class of objects, which may be used as primitive data type.

Notice that the names of most of the nine data types each start with a lowercase letter while the “*S*” in *String* is capitalized. This is because the names of classes in Java are usually capitalized, while the names of primitive data types are not capitalized. We will use Strings starting in this chapter, then learn more about the *String* class in subsequent chapters.



Variables, Constants, and Literals

Data may be represented in Java code as *variables*, *constants* or *literals*.

A **variable** is a memory location holding a specific value, similar to a variable in algebra. A variable has a symbolic name in Java source code, which refers to the value stored in memory. The value, or the link between the variable name and the stored value, can be changed, hence the name *variable*.

A **constant** refers to a value stored in memory which, once initialized, cannot be changed, hence the name *constant*.

A **literal** is a source code representation of a specific value of a built-in data type. The phrase “Hello World!” used in *Lab 1* in *Chapter 1* is an example of a String literal.

Java’s built-in data types are described briefly in the following sections. For more detailed information about Java data types, see the *data types* section of the Java online tutorial at:

<http://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html>

Boolean Data

Java’s boolean data type is named “*boolean*” with a lowercase ‘b’ while the general term is *Boolean*, with an uppercase “B”, named after George Boole. A **boolean** variable may hold the values true or false used to keep track of whether something is true or false. It has the smallest range of any data type.

The answer to any *true* or *false* question, or the equivalent, such as *yes* corresponding to *true* and *no* corresponding to *false*, can be stored as boolean data. A checkbox on a job application keeping track of whether or not the applicant has a driver’s license, for example, can be coded in Java as boolean data.

PENNSYLVANIA VOTER REGISTRATION APPLICATION
DO NOT WRITE IN SHADED AREAS

1 Are you a citizen of the United States of America? ☐ Yes ☐ No
 Will you be 18 years of age on or before election day? ☐ Yes ☐ No

2 ☐ New Registration ☐ Change of Name ☐ Change of Address ☐ Change of Party ☐ I am a Federal or State employee and wish to retain my voting residence in the county where I last resided.

3 Mr. Last Name First Name Middle Name/Initial Jr. Sr. II III IV DL # OR SS# (last 4 digits) Telephone Number (Optional)

4 Address of residence, include street and city (Use zip code) Street number or name) (If only P.O. box, see above) Apt # State Zip Code

5 Municipality where you live County where you live Mailing address (if different than address of residence) City State Zip Code

6 Date of Birth 7 Race (Optional) 8 Previous registration 9 In which party do you wish to register?
☐ Democratic ☐ No affiliation ☐ Republican ☐ Other (Please specify)

10 Address of previous registration 11 Year of previous registration

Are you a citizen of the United States of America? ☐ Yes ☐ No
 Will you be 18 years of age on or before election day? ☐ Yes ☐ No

Figure 2 –
Boolean questions

Figure 2 shows two questions from the Pennsylvania Voter Registration Form that each are Boolean questions – they have yes or no answers that can correspond to the boolean values *true* or *false*.

Boolean data can be represented by a single bit of storage space, but often computers use a byte to speed up software since most memory is byte addressable. **Byte addressable** means that each byte of memory has its own memory address, making it faster for the computer to address an entire byte than to address a bit within a byte. Each machine's JVM handles Boolean data according to the rules of the host system. Some use only a single bit to store a boolean value, but most use an entire byte.

There are only two literals for boolean data – *true* and *false*, lowercase with no quotes.

Text Data

Text data is made up of characters. A **character** is a single symbol from the alphabet of a language. In English, the name "John", for example, is made up of four characters – *J*, *o*, *h*, and *n*.

Unicode and the older **ASCII** character set (American Standard Code for Information Interchange) are the most commonly used codes for storing text data. Java stores characters using **UTF-16**, a 16-bit version of Unicode. This means that each Java character is 16 bits (two bytes) long. See The Unicode Consortium Website at: <http://www.unicode.org/> for more information.

Java has two built-in data types for storing text data:

- **char** – *char* is a primitive Java data type used to store a single UTF-16 character. The *char* data type has a length of 16 bits (two bytes). Char literals are enclosed in single quotes – 'X'.
- **String** – a Java String is a set of Unicode characters stored as an instance of the String class of objects. Special support is built into Java for String objects. 16 bits, (two bytes) are needed for each character in a Java String. Notice that the names of the primitive data types all start with a lowercase letter, while *String* starts with an uppercase 'S' because it is the name of a class of objects and class names are capitalized.

A **String literal** is a string of characters in quotes. It can be used anywhere a String value or variable can be used in Java, such as in output statements. "Hello World!" is an example. Strings literals are indicated by enclosing them in double quotes – "name".

Any UTF-16 character can be used as a Java character or in a Java String. The term *alphabetic character* refers to upper and lowercase letters, usually in reference to the Latin alphabet used for English. The term *numeric character* refers to decimal numeric digits 0 through 9. The term **alphanumeric character** refers to the two together – alphabetic characters and numeric characters but it does not include *special characters*, such as \$, !, ? or #. All characters that are not decimal numeric digits 0 through 9 are called **non-numeric characters**.

The *String* and *char* data types are used for information that is not normally used to perform arithmetic. Names, labels, and messages are most often stored using the *String* and *char* data types. This includes ID numbers used as labels identifying people or things, such as Social Security numbers, student numbers, and product serial numbers. Even though we call them numbers, they are really used as text data and often include non-numeric characters. We also don't want them to be rounded off or truncated, as numbers sometimes are.

$$X = \sqrt{\text{zip code} + \frac{\text{social security number}}{2}} ?$$

Representing ID Numbers

ID numbers are numbers that identify a person, place or thing, such as *social security number* or *zip code*. They are not stored as numeric data because they will not normally be used for arithmetic. (*When is the last time you needed average student number, or the square root of your zip code?*)

They are really labels, and as such are *Strings*. In fact they should be Strings so they can include non-numeric characters, and so they aren't rounded off, as some numeric occasionally is.

Special Characters in Strings



Escape sequences are codes for special characters. They are called escape sequences because they originally involved the use of the escape key on computer terminals. They were defined as part of the ASCII code and are now part of Unicode. They can be used as text data in Java and most other modern programming languages. Many are *control codes*, which do not print a character but control the way characters are printed.

Each Unicode character has a code number, usually expressed as a hexadecimal number. A character can be included in a Java String literal by typing the backslash followed by its escape sequence character, or by typing a backslash followed by a *u*, followed by its Unicode hexadecimal code number. For example, this Java statement prints a String starting with a *tab* and ending with an *alpha*:

```
System.out.println("\t This prints a Greek alpha \u03b1.");
```

Here are examples of some Unicode control characters with escape sequences and Unicode non-Latin printable characters. All can be represented using their UTF-16 values.

Control Codes (<i>escape codes</i>)		
name	sequence	Unicode (hexadecimal)
Backspace	\b	0008
Tab	\t	0009
Linefeed (new line)	\n	000a
Form feed (new page)	\f	000c
Double quote	\"	0022
Single quote	\'	0027
Backslash	\\	005c
Printable Characters (<i>non-Latin</i>)		
name	character	Unicode (hexadecimal)
Greek uppercase theta	Θ	0398
Greek lowercase alpha	α	03b1
Greek lowercase pi	π	03c0
Hebrew alef	א	05d0
Arabic alef	ﺀ	0627
pointing hand	☞	261b
smiley face	😊	263a

There are Unicode UTF-16 characters from many alphabets (Arabic, Cyrillic, Greek, Hebrew, etc.) and for many special characters. The null value is Unicode 0000, which is distinguished from a blank space (0020), or a zero digit (0030). The non-printing control characters are used for special purposes, such as in data communications and to format output.

For more information, see: The Unicode Consortium Website at: <http://www.unicode.org>

and the interactive Unicode Character Code pages online at: <http://www.unicode.org/charts>

Integer Data

Java has four primitive integer data types: *byte*, *short*, *int*, and *long*:

- A **byte** is an 8-bit value representing a signed integer with a minimum value of -128 and a maximum value of +127. It is the shortest of all integer data types, using only a single byte (8 bits) for each value.
- A **short** integer has a minimum value of -32,768 and a maximum value of +32,767. It requires 16 bits of memory (two bytes) to store each value.
- **int** data has a minimum value of -2,147,483,648 and a maximum value of 2,147,483,647, using 32-bits of memory (four bytes). It is the most commonly used integer data type.
- A **long** integer has a minimum value of -9,223,372,036,854,775,808 and a maximum value of 9,223,372,036,854,775,807 (Roughly $\pm 9.2 \times 10^{18}$). It is the longest primitive integer data type, using 64 bits of memory (eight bytes) for each value.

Integers are entered in Java by typing the digits, with no commas for thousands, etc. A negative sign should be used for negative numbers. All primitive integers in Java are stored using two's complement arithmetic, studied in more detail in computer-related Math courses, such as Math 121 or Math 163 at Community College of Philadelphia. Java does not have primitive data types for unsigned integers, as some languages, such as C++, do. The only difference between any two Java integer data types is the length of the number, as determined by its storage space.

The most commonly used integer data type is *int*. The *short* or *byte data type* can be used to save memory, such as in an embedded processor in a small chip with little memory. The *long* data type can be used when integers beyond the range of the *int* data type are needed.

Floating Point Data

Java's floating point numbers are stored using a **mantissa-exponent format** similar to scientific notation with a mantissa, an exponent, and two signs, one for the mantissa and one for the exponent. For example, the value **437.65**, which would be **4.3765 x 10²** in base-ten scientific notation, would be stored as **+4.3765E+02** as a Java floating point number. The value before the E is the number's mantissa and the value after the E is its exponent.

It has become customary in Computer Science to refer to the mantissa as the **significand** in a floating point number to distinguish it from the mantissa in a logarithm. Floating point numbers are not stored in logarithmic formats, they are stored in a log-linear format, so what is called the mantissa in a floating point number is not the same thing as a mantissa in a number stored in a logarithmic format.

Floating point number formats in Java are compliant with the IEEE's *Standard for Binary Floating-Point Arithmetic* (IEEE 754-2008).

IEEE Floating Point Numbers

The IEEE's *Standard for Floating-Point Arithmetic* (IEEE 754) defines formats for floating point numbers and rules for operations on those numbers. Java uses two binary formats from the standard for binary floating point numbers: the 32-bit binary format for the *float* data type, and the 64-bit binary format for the *double* data type. Values are stored in a binary format but displayed by default in a decimal format.

Floating point numbers are similar to scientific notation:

+4.3765E+02 is **4.3765 x 10²**, four point three seven six five times ten to the second power.

An IEEE floating point number has:

- a **sign** indicating if the value is positive or negative.
- a **significand** containing the significant digits of the number. The significand is more formally known as the *mantissa* of a floating point number, but is often called a significand to distinguish it from the mantissa in a logarithm. Floating point numbers are in a log-linear format, not a logarithmic format.
- an **exponent**. IEEE 754 uses an offset for the exponent which accounts for the sign, so the sign of the exponent is not stored separately.
- The base, which is always 10, is not stored.

32-bit format used for float values

Sign	Exponent	Significand
1 bit	8 bits	23 bits

The 23-bit **significand** is always accurate to at least six decimal digits, sometimes seven, depending on the exact value. It is safe to rely on six digits of accuracy.

The 8-bit **exponent** is in the binary range of -127 to +127, roughly equivalent to the decimal range 10^{-38} to 10^{+38} . ($2^{127} \approx 10^{38}$)

64-bit format used for double values

sign	Exponent	Significand
1 bit	11 bits	52 bits

The 52-bit **significand** is always accurate to at least 15 decimal digits, sometimes 16, depending on the exact value. It is safe to rely on 15 digits.

The 11-bit **exponent** is in the binary range of -1023 to +1023, roughly equivalent to the decimal range 10^{-308} to 10^{+308} . ($2^{1023} \approx 10^{308}$)

Java floating point formats are based on the 1985 standards (IEEE754-1985). The standard was expanded in 2008 (IEEE754-2008) and adopted by the ISO. The old formats are a subset of the new definition, and floating point values in Java are compliant with the newer standard.

There are two primitive data types in Java for floating point numbers:

- **float** – a single precision floating point value that is accurate to six decimal digits, with values ranging from roughly 10^{-38} to 10^{+38} . A float requires 32-bits of storage space (four bytes).
- **double** – a double precision floating point value that is accurate to fifteen decimal digits, with values ranging from roughly 10^{-308} to 10^{+308} . A double requires 64-bits of storage space (eight bytes). The name double comes from the fact that it uses double the memory, not that the mantissa or exponent are double the accuracy.

Floating Point Numbers and Scientific Notation

+ 4.3765 E + 02

value of the exponent
sign of the exponent (big or small value)
value of the significant (also called mantissa)
sign of the significant (positive or negative)

some examples:

$+4.3765E+02$	$= 4.3765 \times 10^2$	$= 435.765$
$4.3765E+03$	$= 4.3765 \times 10^3$	$= 4,357.65$
$1.25E-01$	$= -1.25 \times 10^{-1}$	$= 0.125$
$-1.80E+02$	$= -1.8 \times 10^2$	$= 180.0$

Floating point numbers may be typed in Java as decimal numbers, such as 1234.56, or in their significant-exponent format: +1.23456E+06. The significant is normalized, which means there is only one significant digit before the decimal point. 45.654E+02 would be normalized to 4.5654E+03. Floating point numbers all have decimal points. 0.0 is a floating point value. 0 is an integer value.

Checkpoint 2.1

1. What are Java's nine built-in data types and how do they fit into four major categories?
2. Which Java built-in data type is not a primitive data type?
3. How do each of Java's integer data types differ from one another?
4. What is a literal and how are literals represented for different built-in data types?
5. Describe the formats used to store floating point data in memory.

Note: Be careful about answers that are correct, but incomplete. For example, *float* and *double* are the names of the formats used for floating point data, but questions 5 asks for more than just their names.

Numbers from the Real (and Imaginary) World

We've looked at integer and floating point numbers in Java, but what about other kinds of numbers, such as whole numbers, real numbers, and so on? It turns that some are well-defined and some are subject to interpretation.

An integer is numbers without a fraction {..., -3, -2, -1, 0, 1, 2, 3, ...}. Any of Java's integer datatypes can be used to represent integers, within the range of values for each type.

The terms **whole numbers**, **natural numbers** and **counting numbers** are without universally agreed upon definitions. Some mathematicians say *whole numbers* and *integer* mean the same things, while others say *whole numbers* start at zero. Some say *natural numbers* start at zero and include only the positive integers. Others say *natural numbers* start at one. The same is true for *counting numbers* – some say they start at zero, some say at one.

Mathematicians and computer scientists who wish to be precise use the terms *positive integers* and *non-negative integers* to define two subsets of integers: the **positive integers** start at one {1,2,3,...}, while the **non-negative integers** start at zero {0,1,2,...}. The non-negative are also called **unsigned integers**. Java does not have a special data type for unsigned integers as some languages do.

A **rational number** is any number that can be represented by a ratio of integers; basically a fraction whose numerator (top) and denominator (bottom) are both integers. (The denominator cannot be zero.) Integers are a subset of the rational numbers.

A **real number** is any number that can be represented on a number line. Basically, if we can draw a line of a certain length, even if the length cannot be expressed exactly as a rational number, it is a real number. The hypotenuse of a right triangle with two sides each 1 unit long is $\sqrt{2}$, which is a real number that does not exactly equal any rational number. Such numbers are known as *irrational numbers*. An **irrational number** is any real number that cannot be represented exactly by a ratio of integers, such as π or $\sqrt{2}$.

We cannot represent all real numbers in Java, but we can represent a decimal approximation of real numbers using floating point numbers. For example, π is 3.14159265358979 as the 15 digit double value +3.14159265358979E+00, which is accurate enough to calculate the radius of a sphere the size of the Earth within one-millionth of an inch.

Numbers that involve negative square roots are **imaginary numbers**, since negative values have no square root. **Complex numbers** combine real and imaginary numbers. They are important in fields such as electronics. Basically, if you need to use them you know what they are, if not, then don't worry about it for now. The Apache Commons website has a complex number API used by scientists and engineers. (See: <http://commons.apache.org/proper/commons-math/userguide/complex.html>)

Lesson 2.2 Variables and Constants

The variables and constants mentioned in the discussion of data types are an essential element of computer programming.

In addition to a data type, each variable and constant in Java has a *scope of existence*. The *scope of existence*, or simply the **scope**, of a variable or constant is that part of a program in which it exists and can be used.

Variables may be local variables within a method or they may be used to refer to the properties of objects, such as the color of a car or the name of a person. In the rest of this section we will focus on local variables. The properties of objects are discussed in later chapters, but for now you should know that they also have a name and a data type, just as local variables do.

Local Variables

The variables in Java methods are called **local variables**. The scope of a local variable is determined by its *lexical context*, which means it depends on the variable's place within specific language. A local variable can only be used within the block of code in which it is declared. The beginning and end of Java blocks of code are marked by braces { ... } as we saw in Lab 1 chapter 1. A method is a block of code, since it begins and ends with braces. We may also have blocks within blocks, such as a block of code to be repeated in a loop, or a block of code to be executed by an *if* command.

A variable only exists and can only be used in the block of code in which it is declared. A variable declared inside one method does not exist in another method and cannot be used in another method. Another method could have a different variable with the same name, but they would be different variables, referring to different memory locations.

To declare a variable means to state the name and data type of a variable, and possibly give it an initial value. Here are three examples:

```
int sum;                // the sum of the test scores
double angle = 47.63;   // angle of elevation
String zipcode;         // the customer's zipcode
```

In the first example, the variable named *sum* is declared to be an *int* variable. It has no initial value. A variable without a value is a **null variable** – it is not equal to zero, it has no value. There is a difference.

In the second example, *angle* is declared to be a variable of type *double*, with an initial value of 47.63. All variables for primitive data types and Strings can be initialized in this way when they are declared. The value should be a literal of the same data type as the variable, or an expression yielding a value of the correct data type.

In the third example, the variable *zipcode* is declared as a *String* variable. Notice that the name *zipcode* is lowercase, as are almost all variable names. *String* with an uppercase 'S' is the name of a class; *zipcode* refers to an instance of the *String* class. In this example, *zipcode* is not initialized, so it will have a null value until it is used in the program.

The inline comments in the variable declarations above make it easier for people reading the code to understand it – including the code's author, reading the code months or years after it was written.

Variable declarations may appear almost anywhere in a Java program, but it is a good programming practice to declare all variables for a method at the beginning of the method with a comment describing each variable. This practice, known as **front loading declarations**, does two things:

1. It creates a *data dictionary* at the beginning of a method, making it easier for people who read the code to find out what variables are used in the code and what they represent.
2. It makes it easier to read a program. Data declarations in the middle of code can sometimes make the code harder to follow, in a subtle way.

Constant Declarations

A constant in Java is declared and used just as a variable is, except that its value cannot be changed. We designate a constant by putting the keyword **final** in front of the type in its declaration, as in these examples:

```
final double PI = 3.141592653589793;  
final String PAGENOTE = "© 2014 – All rights reserved";  
final int CLASS_SIZE_LIMIT = 36;
```

Notice that constant names are all uppercase, with an underscore character separating parts of a compound name. This is a Java programming convention.

Hardcoding

Putting a data value directly into your code is known as **hardcoding** data, and is generally frowned upon by good programmers (and instructors who will be grading your assignments).

It is a bad habit to develop, because it can lead to errors and inefficiency. Imagine that you will use 365.25 as the number of days in a year many places in a program. Instead of typing 365.25 many times or copying and pasting it many times, you can declare a constant `DAYS_PER_YEAR`, and just use it each time. The same holds true for String values used repeatedly like “*Northwest Philadelphia Regional Center*” in CCP software, or “*Aleksandr Isayevich Solzhenitsyn*” in an application about Russian dissidents who won the Nobel Prize for literature in 1970.

It is good programming practice to develop the habit of using constants instead of hardcoding values in software.



Variable and Constant Names – Java Identifiers

A **Java identifier** is a string of Unicode characters that identifies something in Java code. Identifiers are used to name variables, methods, classes, and so on. The *Java Language Specification (JLS)*¹ section 6.2 describes the acceptable syntax of a Java identifier and also suggests programmers follow common naming conventions.

¹ Available online from Oracle at: <http://docs.oracle.com/javase/specs/jls/se8/html/jls-6.html#jls-6.2>

According to the JLS, Java identifiers must start with a letter of the Latin alphabet (a-z), an underscore character (Unicode 005F), or a dollar sign (Unicode 0024). The use of the underscore and dollar sign are for historic reasons, and the JLS discourages their use as the first character in an identifier in modern code. The remaining characters in an identifier may also include decimal numeric digits (0-9), so they may be any the *alphanumeric* characters or the underscore or dollar sign.

There are two forms of names in Java, *simple names* and *qualified names*. A **simple name** is a single identifier. A **qualified name** is a sequence of two or more simple names separated by periods. Qualified names identify the context in which a simple name is used. For example, a property *lastName* for an instance of the Student class named *student1* would have the qualified name *student1.lastName*.

Simple names are used most of the time for local variables in Java code. Qualified names are used when the context needs to be specified, such as when code refers to properties or methods from a class.

There are four primary naming conventions in Java:

- names should be meaningful. Minimize the use of *x*, *y*, *a*, *b*, or things like *num1* or *num2*.
- class names are in Upper CamelCase, such as *HelloWorldApp*, *Math*, or *Student*.
- variable and method names are in lower camelCase, such as *sum*, *realRoot1*, or *hourlyRate*
- constants are named in all UPPERCASE, using an underscore to separate part of a compound name, such as *PI*, *GROWTH_RATE*, or *MAX_WEIGHT*. This clearly distinguishes constants.

Java **Keywords** are part of the language and are meaningful to the compiler. They may not be used as identifiers in Java. Some examples are *int*, *class*, *double* and *if*. Keywords are also called *reserved words*.

Java has a relatively small set of 50 keywords:

abstract	continue	for	new	switch
assert	default	if	package	synchronized
boolean	do	goto	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

By comparison, C++ has 86, C# has 79, Swift has 71, Python 2.0 had only 31 and COBOL has over 400.

Checkpoint 2.2

1. What determines the scope of a local variable?
2. What is the difference between the way constants and variables are declared?
3. What is hardcoding and why is it better to use constants than to hardcode data?
4. What is a qualified name and why does it have more than one part?
5. What are the benefits of frontloading variable and constant declarations?

Lesson 2.3 Assignment Statements and Expressions

An **assignment statement** assigns a new value to an existing variable. It has two parts, separated by an equal sign:

```
variable = expression;
```

The only thing to the left of the equal sign is the name of the variable. An expression describing the value to be assigned is to the right of the equal sign.

Here are three examples:

```
angle = 45;           // int assignment, hardcoded
grossPay = hours * rate; // math expression
city = "Philadelphia"; // String assignment
```

Expressions tell a computer how to determine or calculate a value. Numeric expressions often look like expressions in elementary algebra and most of the rules are the same as in elementary algebra. The computer will attempt to resolve the expression to end up with a single value of the required data type, then assign that value to the variable.

An expression should yield a value of the same data type as the variable to which it is assigned. If not, type casting may occur. Type casting is discussed in more detail later in this section.

Expressions have **operands**, which are the values used in an expression, and **operators**, which are symbols or functions indicating the operations to be performed in an expression. In the expression $sum = a + b$, the terms sum , a , and b are the operands, while the plus sign is the operator indicating the addition operation.

Instructions included as part of a data type tell the computer how to perform basic operations on that type of data, such as addition, subtraction, multiplication, and division for integer data. The symbols used in assignments statements indicate these operations. **Functions** are methods that return a value, such as a math function to return the square root of a number or a String function to convert a String to all uppercase characters.

The nature of an operation depends on the data types of the operands being used with the operation. For example, The Math library's round function can be used to round off a double or float to the nearest integer, but there is no need for such an operation for Strings:

```
x = Math.round(17.6) results in x being set to 17.
```

```
x = Math.round("Hello world!") doesn't work, because "Hello world!" is a String, not a number,
and the round operation is not defined for Strings.
```

Polymorphism

Sometimes a symbol for an operation or the name of a method in a method call can trigger one operation for one data type and another operation for a different data type. In computer programming, this is a form of **polymorphism**, which means the same name or symbol can stand for different operations on different data types. The meaning of the plus sign $+$ depends on the data types of its operands. The plus sign is polymorphic because it performs different operations on different types of data:

- The Java instruction `System.out.print(12 + 16)` prints the number 28. The operands 12 and 16 are int values, so the computer will perform the operation associated with the plus sign for integers — simple integer addition.
- The Java instruction `System.out.print("Joe" + "Smith")` prints the String "JoeSmith". "Joe" and "Smith" are Strings, so the computer will perform the operation associated with String data — **concatenation**, sticking the two strings together to form a new longer string.

The plus sign signifies addition for int values but concatenation for Strings. It has multiple meanings, with its meaning bound to the data types of its operands. This is an example of what is known as **ad hoc polymorphism**, in which the meaning of symbol or a method call depends on the data type of its operands. Ad hoc polymorphism is also called **operator overloading** and **function overloading**.

Polymorphism is studied in more detail in CSCI 112 and 211, where students learn about **parametric polymorphism**, which means a generic method has been written to work with any data type. Ad hoc polymorphism is limited to a few specific data types, while parametric polymorphism will work with any data type. For now, it is enough to have a general idea of what polymorphism is – the same method name or symbol can trigger different operations for different data types. Polymorphism is one of the key aspects of object-oriented programming.

Arithmetic Operations

Java has several built-in arithmetic operations that can be performed on integer and floating point data:

- | | | |
|--|----|--------------------------------|
| • Addition | + | (the plus sign) |
| • Subtraction | - | (the minus sign) |
| • Multiplication | * | (the asterisk) |
| • Division | / | (the slash) |
| • The remainder operation | % | (the percent sign) |
| • Unary negation | - | (the minus sign) |
| • Unary increment (prefix and postfix) | ++ | (two plus signs with no space) |
| • Unary decrement (prefix and postfix) | -- | (two minus with no space) |

Addition, subtraction, multiplication and division are similar to the same operations in elementary algebra. Each of these operations takes two operands.

```
sum = addend1 + addend2;
difference = minuend - subtrahend;
product = multiplier * multiplicand;
quotient = dividend / divisor;
```

Java does not understand implied multiplication, $z = 3x + y$ must be written as:

$$z = 3 * x + y;$$

Quotients and Remainders

A quotient from integer division will be an integer, the remainder is lost. For example, 20 divided by 3 is 6 remainder 2. The result will be the quotient, 6. The remainder of 2 is lost. The **modulus operation**, also called the **remainder operation** returns the remainder, not the quotient. The remainder is called the *modulo* of a number (17 modulo 5 = 2). The operator is the percent sign instead of the slash:

```
Remainder = dividend % divisor;
```

This is read as “Remainder equals dividend modulo divisor.”

Here is an example of how the modulus can be useful. Let’s assume that we have 20 feet and want to know how many yards there are in twenty feet and how many feet will be left over. There are three feet in a yard. 20 divided by 3 is 6, remainder 2. Division tells us how many whole yards, 6. Modulus tells us how many feet will be remaining, 2. In general, we could use two instructions in a method like the one below to capture quotient and remainder, assuming the operands are integers:

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

    int oldFeet;                // the initial distance in feet;
    int yardsInOldFeet;         // the yards in the distance
    int feetLeftOver;           // the number of feet left over
    final int FEET_IN_YARD = 3; // constant 3 ft. = 1 yd.

    // insert some code here to get old feet from the user

    yardsInOldFeet = oldFeet / FEET_IN_YARD; // calculate yards
    feetLeftOver = oldFeet % FEET_IN_YARD;    // remaining feet

    // inset code here to beautifully display the results
} // end main()
```

Figure 3
calculating quotients
and remainders

The result of floating point division is a floating point value: $20 / 6 = 3.33333$. The modulus operation may also be used to return the remainder from floating point data. $20.0 \% 6.0 = 2.0$.

Unary Operations

Several of the operations on integers are unary operations. A **unary operation** only has one operand. *Negation*, *increment* and *decrement* are unary operations.

Negation means reversing the sign of a number; a positive value becomes negative and a negative value becomes positive. We do this by simply putting a minus sign immediately in front of an integer value. For clarity, the value might be placed in parentheses, like this:

```
loss = -(gain);
```

If *gain* is positive, *loss* will be negative. If *gain* is negative, *loss* will be positive.

The **Increment** operation increases the value of an integer or floating point value by one. If *count* is ten, then incrementing *count* makes it eleven. The increment operator is a double plus sign.

Increment can be used alone as an instruction, as in this example:

```
count++; // increment count (add 1 to count, equivalent to count = count+1)
```

The new value of count will be the old value of count plus one more.

The increment operation can also be used in an expression, such as:

```
total = count++;    // total = count, then increment count
```

If the operator is after the variable this is called a *postfix increment*. A **Postfix Operation** occurs after the variable is used. The value of *count* is assigned to *total*, then *count* is incremented. If the old value of *count* was 10, then after the instruction, *total* will equal 10, *count* will equal 11.

Increment can also be a *prefix operation*. A **Prefix Operation** occurs before the variable is used. Like this:

```
total = ++count;    // increment count, then assign the value to total
```

In this case, *count* will be incremented first, then the value of *count* is assigned to *total*. If the old value of *count* was 10, then after the instruction, *total* and *count* will both equal 11.

The **decrement** operation decreases the value of an integer or floating point by one. The decrement operator is a double minus sign. It is similar to the *increment operation* except it subtracts instead of adds. It can be used prefix or postfix, just as increment can be.

```
Total = count--;    // total = count, then decrement count
Total = --count;    // decrement count, then assign the value to total
```

Java also has **compound assignment** operation. A **compound assignment** operation combines an arithmetic operation with an assignment statement to add, subtract, multiply, or divide a fixed value with the existing value of the variable, then assign the resulting value to the variable, such as:

```
x += 5;            // compound assignment, equivalent to x = x + 5;
```

How does this work? Consider the statement $x = x + 1$. In Java, the new value of *x* equals the old value of *x* plus one. The expression on the right side of the equals sign is evaluated, then the value is assigned to the variable on the left. This works even if the same variable is on both sides. We have just seen that an increment statement $x++$ can take the place of $x = x + 1$. In a similar manner, $x = x + 5$ can be replaced with $x += 5$.

There are five compound assignment operators:

addition assignment	subtraction assignment	multiplication assignment	division assignment	modulo assignment
+=	-=	*=	/=	%=

Note that there are no spaces between the arithmetic operator and the equals sign in compound assignment statements.

The Java Math Class

The Java programming language supports elementary arithmetic, but to do more advanced mathematical computation, such as working with exponents, logarithms, or trigonometry, Java provides a library of methods in the *Math* class, which is part of the *java.lang* package in the JDK.

The **java.lang package** is a set of classes with features to support Java programming. It is unique because it is the only external package that does not require an import statement. We will learn more about import statements and using other packages later in this chapter.

The **Math class** has a set of methods that perform math functions. The Math class methods are *static* methods, which means they are each invoked using the name of the class as part of their qualified name, not an instance of the class. Square root is an example of this. It has the simple name *sqrt*. Its qualified name includes the class name: *Math.sqrt*.

This table shows some methods available in the *Math* class in *java.lang*. The values in parentheses are the data types of a method's parameters, the values after the method are the type the method returns.

java.lang.Math	
abs(int x): int abs(long x): long abs(float x): float abs(double x): double	the absolute value of x
cos(double x): double	the cosine of x radians
exp(double x): double	e^x where e is 2.71828
hypot(double x, double y): double	square root of ($x^2 + y^2$)
log(double x): double	natural log of x
log10(double x): double	base 10 log of x
max(int x, int y): int max(long x, long y): long max(float x, float y): float max(double x, double y): double	the greater of the two arguments
min(int x, int y): int min(long x, long y): long min(float x, float y): float min(double x, double y): double	the lesser of the two arguments
pow(double x, double y): double	x^y x to the y power
sin(double x): double	the sine of x radians
sqrt(double x): double	the square root of x
tan(double x): double	the tangent of x radians
todegrees(double x): double	converts x degrees to radians
toradians(double x): double	converts x radians to degrees

Notice that several of the functions in the Math class are *polymorphic*, which means they work with more than one data type. Polymorphic functions are actually different functions with the same name, but their parameters have different data types. If two functions have the same name, the compiler will decide which function to use based on the data types of their parameters. Two functions cannot have the same name and identical parameters. If the name is the same, the parameters must be different.

The Math class also has two defined double constants accurate to six decimal places:

- `Math.E` = 2.718281828459045, the closest *double* value Euler's number, the basis for natural logarithms
- `Math.PI` = 3.141592653589793, the closest *double* value to π

For a full description of the Math class, see the oracle Math class reference:

<http://docs.oracle.com/javase/8/docs/api/java/lang/Math.html>

The Math class is also discussed in the Oracle Java tutorials, online at:

<http://docs.oracle.com/javase/tutorial/java/data/beyondmath.html>

Here is an example showing how to use the square root function from the Math class in a Java method:

```
public static void main(String[] args) {  
    double squareArea;    // the area of a square  
    double side;          // the length of the side of the square  
    squareArea = 100.0;   // 100.0 is floating point  
  
    side = Math.sqrt(squareArea); // uses the square root function  
  
    // output result – Strings and variables concatenated into one String  
    System.out.println("The side of a square with area " + squareArea + " is " + side);  
  
} // end main()
```

Figure 4
using Java's Math class

The output looks like this:

```
The side of a square with area 100.0 is 10.0
```

The variables in this method are both double. To keep the example simple, *squareArea* is simply set to 100.0. (Normally this value would come from somewhere else, such as user input.) Notice that it is 100.0 instead of 100 to avoid type casting from an integer value to a floating point value.

The method *sqrt*, which calculates a square root, is invoked from the Math library in the statement

```
side = Math.sqrt(squareArea);
```

Math is the name of the class, and *sqrt* is the name of the method. We use the qualified name *Math.sqrt* to access a method in another class. *squareArea* is an argument of the *sqrt* function, or in terms of Java, a parameter that is passed to the method. The method will use the value of *squareArea* to calculate a square root and then return the value of the square root, which will be used in our expression. In this case, it is the only value in the expression, so *side* is set to the value returned by the *sqrt()* function.

Order of Operations

The order of operations in a programming language is affected by three things:

- order of evaluation
- operator precedence
- grouping symbols 610 992 5620

The **order of evaluation** for a language is basically the direction of the language – left to right for Java, just as it is for English. (*Hebrew and Arabic are examples of languages that are read from right to left, while some Asian languages are read from top to bottom.*)

Order of evaluation is superseded by operator precedence. **Operator precedence** is the precedence given to one operator over another by a compiler as it evaluates expressions.

Here is part of the table of operator precedence from Oracle’s Java Documentation:

Operator Precedence	
unary postfix	<code>expr++ expr--</code>
unary prefix	<code>++expr --expr -expr</code>
multiplicative	<code>* / %</code>
additive	<code>+ -</code>

The table above shows us that for elementary math operations Java will evaluate multiplication and division before addition and subtraction, but that multiplication and division have the same precedence, and addition and subtraction have the same preference. Does *MDAS* sound familiar? It is the same order of operations for elementary math in elementary algebra.

Here is an example:

`x = 20 + 4 / 2;` What does x equal? Normally we perform operations in order from left to right, but division has precedence over addition, so we should do the division first. The correct answer, in elementary algebra and in Java is `x = 22`, not `x = 12`.

We can explicitly change the order of operations by using parentheses, as in this example:

`x = (20 + 4) / 2;` In this case, `X = 12`, not `22`. The parentheses tell the computer to evaluate the addition before the division.

Parenthesis are a grouping symbol in math. We can always use parentheses to explicitly tell the computer to evaluate one operation before another.

Parentheses are also very important when translating fractions into a programming language. In elementary algebra, the fraction bar is a grouping symbol just as parentheses are grouping symbols. Consider the following example:

$$x = \frac{3 + 9}{3 - 1}$$

What does x equal? $3 + 9 = 12$, $3 - 1 = 2$, and $\frac{12}{2}$ is 6. The fraction bar tells us to evaluate the terms in the numerator and denominator before dividing.

But what happens if we translate this into Java? Many people would simply enter:

`x = 3 + 9 / 3 - 1;`

However this is not correct, because division has precedence over addition and subtraction, yielding:

$$3 + \frac{9}{3} - 1, \text{ which is } 3 + 3 - 1, \text{ which is } 5.$$

The correct form of $x = \frac{3+9}{3-1}$ translated into Java would be:

```
x = ( 3 + 9 ) / ( 3 - 1 );
```

This example shows us that parentheses should be placed around the numerator and the denominators of fractions with operations in them when we are translating the fractions into expressions in Java. In general, if you are not sure what the computer will do first, you can tell the computer what you want it to do first by using parentheses.

Note that functions with expressions in the argument of the function will resolve the argument to a single value before invoking the function. For example, in `x = Math.sqrt(4 + 20);` The computer will evaluate `4 + 20` to a single value before it uses the square root function.

Commutative and Associative Behavior

Most Java arithmetic operations are commutative and associative in the same manner as their counterparts in elementary algebra. **Commutative** means two operands can exchange positions and the result of the operation is the same; in other words, the order of the operands does not affect the result. ($A+B = B+A$)

Addition and multiplication are commutative, subtraction and division are not. The order of the operands in subtraction and division (including the modulus operation) affects the results.

Even when the operation is commutative, computer programmers should still try to be consistent in the order in which they use operands. This is related to Crewton Ramone's *corpulent midget rule*.² Carpenters always specify length, then width, then height when listing dimensions. Even though $W \times L$ is the same as $L \times W$, if a carpenter gets the numbers in the wrong order on a job site, we could end up with a door for corpulent midgets (3 feet high and 7 feet wide instead of 7 feet high and 3 feet wide). The order of two values might make a difference in how a person interprets the data, even when it doesn't make a difference in the result of a commutative operation.

Associative means that if the same operation is used several times in a row, such as $A+B+C$, then it does not matter which operation is performed first: $(A+B)+C = A+(B+C)$. Java is *left associative*, meaning that it will perform $A+B$ then add the result to C , but if $B+C$ were added to A , the result would be the same.

Addition and multiplication are associative, but subtraction and division are not.

² Crewton Ramone is a math educator in Hawaii. See *Crewton Ramone's House of Math*, online at: <http://www.crewtonramoneshouseofmath.com/>

Type Casting

Type casting is the conversion of data from one type to another, according to rules embedded in the compiler. Unnecessary casting should be avoided, because it wastes time during the execution of a program, and could lead to errors. The following code shows examples of type casting:

```
double base = 6.3;
double bonus;
int result;
...
bonus = 2 * base;    // 2 is an int value – type casting will occur
bonus = 2.0 * base; // 2.0 is a double value – no type casting occurs

result = base;      // result is int, base is double. the fractional part of base will be lost.
```

In the first two assignment statements, if 2 is used instead of 2.0, the expression casts the type from int to double since *bonus* and *base* are double and 2 is an int literal. It is better to use 2.0, a double literal. In the third assignment statement, a double value is being assigned to an int variable. In this case, *result* will equal 6, and a loss of accuracy occurs.

If operands are of different data types, then the computer will attempt to resolve the expression using conversions by type casting. In general:

- *int* values can be cast to *double* values with no loss of accuracy (only a loss of time). The *int* value 137 can become the *float* value 1.37E+02 (equal to 137.0) with no loss of information.
- all floating point values cast to any integer type will be truncated, with a loss of information. The fractional portion after the decimal point in the regular (not scientific notation) equivalent of the value will be lost. The *double* value 4.13297E+03 (= 4132.97) becomes the *int* value 4132.
- both integers and floating point values can be cast as Strings, in which case they will become the String expression of the characters in the decimal value. 10.75, becomes the String "10.75". The String is the set of characters '1', '0', '.', '7', and '5', not a numeric value. Often this happens in console or file output. Since I/O operations are slow in any case, casting in I/O operations is usually not a big concern in examining the efficiency of a program outputting data.

Casting Strings to integers or floating point values can be tricky. There are special methods in the String library to parse numeric values from Strings, which we will study in another section.

Converting a value from a data type of greater accuracy and range to a data type of lesser accuracy and range is known as a **narrowing primitive conversion**.

The following are the narrowing primitive conversions in Java:

- short to byte or char
- char to byte or short
- int to byte, short, or char
- long to byte, short, char, or int
- float to byte, short, char, int, or long
- double to byte, short, char, int, long, or float

According to the JLS³, “A narrowing primitive conversion may lose information about the overall magnitude of a numeric value and may also lose precision and range.”

Data errors caused by a loss of information from a type conversion, or by mixing up units (such as feet and meters), or by misplaced minus signs, could have serious consequences. Computers are used in important situations – medical systems, antilock braking systems in cars, passenger airplane controls, the military, and so on. We can help save time, save money, and even save lives by minimizing the chances for data errors in software we create.

Cast Operators

A programmer can force type casting by including a cast operator in an expression before a value to be cast. A **cast operator** is a data type in parentheses explicitly telling the computer to perform type casting, as in this example.

```
int hours;      // hours worked
double rate;    // hourly pay rate
double gross;   // gross pay

... // part of the code here is not shown

gross = (double) hours * rate;
```

In this case, *hours* is cast to a *double* value from an *int* value before the multiplication is performed because *double* is used as a cast operator. This prevents a loss of precision in the multiplication operation.

Checkpoint 2.3

1. What is on each side of the equal sign in a Java assignment statement?
2. What does the term *polymorphism* mean in object oriented programming?
3. What does the % indicate in a math expression and what data types work with this operation?
4. What is the difference between the following two statements:
 - a. `y = x++;`
 - b. `y = ++x;`

5. convert each of the following to Java assignment statements:

- | | |
|---|---|
| a. $sum = \frac{a d + b c}{b d}$ | variables: double sum; int a, b, c, d |
| b. $c = \sqrt{a^2 + b^2}$ | variables: double a,b,c |
| c. $a = \pi r^2$ | variables: double a, r; constant π |
| d. $y = v_0 t \sin(\theta) - \frac{1}{2} g t^2$ | variables double y, v0, t, theta; constant g= 9.8 |
| e. $amount = principal \left(1 + \frac{rate}{n}\right)^{n t}$ | variables: double principal, rate; int n, t |

³ see section 5.1.3 of the JLS, online at: <http://docs.oracle.com/javase/specs/jls/se8/html/jls-5.html#jls-5.1.3>

Lesson 2.4 Data Streams and System Console I/O

In chapter one, we used system console output to display a message on a computer screen. Here we will briefly look at system console I/O in more detail.

System I/O uses data streams, also called I/O streams. A data stream is just a sequence of data flowing from a sender to a receiver. **Input data streams** bring data in from external sources; **output data streams** send data out to external destinations. Java has a variety of packages supporting such I/O.

A **raw data stream** contains unformatted binary data. A **tokenized data stream** contains tokens and delimiters. A **token** is a piece of data. A **delimiter** is a marker that separates one token from another in a data stream. In most modern computers, a delimiter is a Unicode character or a string of Unicode characters. When a person types a list, for example, commas often are used as delimiters.

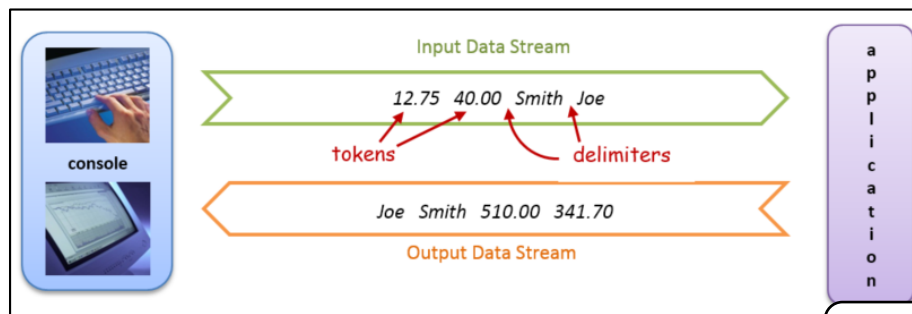


Figure 5
I/O data streams

Java I/O data streams use whitespaces as delimiters. A **whitespace** is:

- a blank space, such as you get by pressing the spacebar (Unicode 0008)
- a set of consecutive blank spaces
- a tab character (Unicode 0009)
- a newline (line feed) character (Unicode 000a)
- a formfeed (page feed) character (Unicode 000c)
- or any of several other technical characters related to file I/O, which we'll see later.

We have already seen the `System.out.print` and `System.out.println` statements for sending output to the system console, which is the system's standard set of input and output devices:

```
System.out.print("The cursor is on this line after print. ");
System.out.println("The cursor is on a new line after println.");
```

A system's standard output device is usually a display screen. *print* and *println* when used with *System.out* usually send unformatted text data to a display screen.

Input is a little tougher because the receiving device for an I/O data stream decides what to do with the data it receives. When we send data out to the console, the operating system decides what to do with the data, so system output is relatively simple, but when data comes into an application, the application needs to decide how to process the data. We need to tell our Java application how to handle the input data stream. There are many ways to do this; one of the simplest is using the `Scanner` class.

The Scanner Class

A **Scanner class** object processes an incoming data stream so that data can be put into variables. There are several input methods in the Scanner class that do this. The methods we will use read tokens of data into variables. The chart below describes some of these methods.

java.util.Scanner	
next(): String	reads the next token and returns it as a String value
nextInt(): int	reads the next token and returns it as a int value
nextLong(): long	reads the next token and returns it as a long value
nextFloat(): float	reads the next token and returns it as a float value
nextDouble(): double	reads the next token and returns it as a double value
nextLine(): String	reads an entire line and returns it as a String value (A line ends with a new line character.)
close()	closes the stream associated with the Scanner object

The blank parentheses in the chart above indicate methods that do not need a parameter. Except for the close method, the methods are each functions that return a value, whose return type is indicated.

To use the Scanner class methods, we must create an instance of the Scanner class by declaring a local variable to be of data type *Scanner*. The local variable represents the instance of the Scanner class in our method. We can then use the methods from the Scanner class in association with that variable.

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

When we use a method from the Scanner class, we use the qualified name of the method, which includes the name of the instance variable and the name of the method.

```
// get the user's name from the keyboard
System.out.println("Please enter your name: ");
name = kb.nextLine();
```

The println command before the *nextline* command prompts the user to enter data. This creates a **prompt and capture** pair of instructions for user input.

The *nextline()* method returns a String, so we assign the value it returns to a String variable. Most methods that capture data from a data stream work this way.

We should also close the Input data stream for this instance when we are done with it.

```
keyboard.close();
```

The following code demonstrates the use of the Scanner class to capture keyboard input. Note that this example uses the variable *keyboard* as the name of a Scanner class object associated with an input stream from the standard system input device (System.in). The name *keyboard* was chosen by the author of the code. It is a variable name that has no special meaning in Java. Many programmers use a simpler name for the same purpose, such as *kb*.

```
/* keyboardTest.java
 * java code for CSCI 111 with a simple keyboard input example.
 * last edited Dec 21, 2014 by C. Herbert
 */

package keyboardtest;

// one of the following import statements is needed to use the Scanner class
import java.util.*;
import java.util.Scanner;

public class KeyboardTest {

    public static void main(String[] args) {

        String name; // String to holds user's name

        // create a new Scanner class object, associated with console input
        Scanner keyboard = new Scanner(System.in);

        // get the user's name from the keyboard
        System.out.print("Please enter your name: " );
        name = keyboard.nextLine();

        // echo the name back to the screen
        System.out.println("Hello " + name);

        // close input stream
        keyboard.close();

    } // end main()
} // end class KeyboardTest
```

Figure 6
console I/O code

Notice the import statements in the code above. An **import statement** tells a Java compiler to add bytecode from outside of the current source code to the bytecode for the software being compiled. This allows the use of external classes and methods in our Java code.

We may import an entire package or import a single class from a package. In this case, we wish to use the Scanner class in the *java.util* utilities package. The first import statement – `import java.util.*;` – imports the entire package. The second import statement – `import java.util.Scanner;` – specifically imports just the Scanner class. Only one of the two is needed, both are included in this code for demonstration purposes. Theoretically, it is more efficient to import only the specific classes that are used rather than importing an entire package. However, most Java compilers are optimizing compilers, which will import the minimum code needed. In this case, the resulting class file is 993 bytes long, with either import statement when compiling this software using Java 8 with NetBeans 8.

Checkpoint 2.4

1. How is a tokenized data stream different from a raw data stream?
2. Describe how to open and close a data stream for console I/O.
3. Which functions from the Scanner class capture tokens into variables of the following types?
 - a. String (single token)
 - b. String (all tokens on one line as one variable)
 - c. int
 - d. double
4. Give an example of an import statement and describe what an import statement does.
5. Give an example of a “prompt and capture pair” of statements for console IO and how it works.

Lesson 2.5 Documentation First Programming

An important aspect of engineering, including software engineering, is captured in the phrase:

“Design it before you try to build it.”

Documentation first programming is a simple design-first approach to software development in which we begin by writing comments to describe what the software should do, then create code to do what the comments say to do.

We start *documentation first programming* by developing an outline of what the software should do from the specifications for the software, then we turn the outline into a set of comments. We then use those comments in a Java development project as the basis for the code needed to implement the software. If necessary, we refine the comments as we go along.

This section contains a simple example of *documentation first programming*. In this example, we wish to create software for a simple road trip calculator that will tell us the average speed and gas mileage for an automobile journey. The specifications call for a program to:

1. get the distance in miles, driving time in hours, and fuel used in gallons during a long car trip. The data will be input by the user.
2. calculate the average speed (miles per hour), and mileage (miles per gallon) for the trip.
3. display the distance, time, average speed, and mileage for the trip.

This is an example of an I-P-O program – Input, Processing, Output – get some input, process the data, output the results. Many short programs fit the I-P-O pattern. It is a simple example of what’s known in software engineering as a *design pattern*.

To create the road trip software using a documentation first approach, we start with an outline:

1. declare variables
2. set up program to read from keyboard
3. get user input
 - a. distance in miles
 - b. driving time in hours
 - c. fuel used in gallons
4. calculate
 - a. average speed (MPH)
 - b. fuel mileage (MPG)
5. output results
 - a. distance and time
 - b. MPH
 - c. MPG

Next, we create a set of Java comments matching the outline, or copy and paste the outline into an IDE, refining it into comments as we go along. The result should look something like this:

```
// Road Trip Calculator
// by C. Herbert for CSCI 111

// declare variables – all double
// distance traveled in miles
// total driving time in hours
// total fuel used in gallons
// mileage – average miles per gallon MPG
// average speed – miles per hour MPH

// declare an instance of Scanner to read the data stream from the keyboard.

// get distance in miles
// get driving time in hours
// get fuel used in gallons

// calculate Fuel mileage (MPG)
// calculate Average speed (MPH)

// print results – distance and time, MPH and MPG

// close the input Stream
```

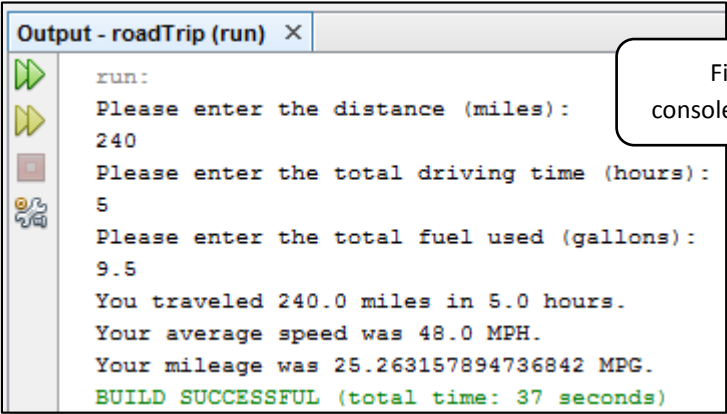
Figure 7
comments describing code

Our next step is to create code to do what each of the comments says to do. Our method might then look something like the code on the next page. An import statement and introductory comments that were not included in the outline have been added to the code.

This approach – starting with the documentation – saves work, makes programming easier to understand, and reduces errors. Comments first help us to design the program, then serve to document what we did after we are finished. This approach separates the process of designing from building, and is a simple way to design something before you try to build it. It is much better than “cowboy coding”, in which we try to design the software as we code and then add comments later.

Many new programming students are tempted to engage in cowboy coding because the first few programs they write are simple and the design is easy, but it is better to develop good programming habits from the beginning. Remember, we’re not here to learn how to write short simple programs, we’re here to learn habits that will serve us well in the long run. Documentation first programming is one step in that direction.

Here is an example of the output from the RoadTrip program showing the dialog:



```
run:
Please enter the distance (miles):
240
Please enter the total driving time (hours):
5
Please enter the total fuel used (gallons):
9.5
You traveled 240.0 miles in 5.0 hours.
Your average speed was 48.0 MPH.
Your mileage was 25.263157894736842 MPG.
BUILD SUCCESSFUL (total time: 37 seconds)
```

Figure 8
console I/O output

```
/* RoadTrip.java
 * program to calculate average speed and mileage for a road trip
 * last edited Dec. 26, 2014 by C. Herbert for CSCI 111
 *

package roadtrip;

import java.util.*;

public class RoadTrip {

    public static void main(String[] args) {

        double distance;    // Distance traveled in miles
        double time;        // Total driving time in
        double fuel;        // total fuel used in gallons
        double mileage;     // mileage - average miles per gallon MPG
        double speed;       // average speed - miles per hour MPH

        // declare an instance of Scanner to read the data stream from the keyboard.
        Scanner keyboard = new Scanner(System.in);

        // get Distance in miles from the keyboard
        System.out.println("Please enter the distance (miles): ");
        distance = keyboard.nextDouble();

        // get Driving time in hours
        System.out.println("Please enter the total driving time (hours): ");
        time = keyboard.nextDouble();

        // get Fuel used in gallons
        System.out.println("Please enter the total fuel used (gallons): ");
        fuel = keyboard.nextDouble();

        // Calculate Fuel mileage (MPG)
        mileage = distance / fuel;

        // calculate Average speed (MPH)
        speed = distance / time;

        // print results - distance and time, MPG and MPH
        System.out.println("You traveled " + distance + " miles in " + time + " hours.");
        System.out.println("Your average speed was " + speed + " MPH.");
        System.out.println("Your mileage was " + mileage + " MPG.");

        // close the input stream
        keyboard.close();

    } // end main()

} // end class
```

Figure 9
the road trip program
chapter file: *RoadTrip.zip*

NetBeans Notes – Formatting Source Code

Java code in a NetBeans project can be reformatted to correct indenting by using the **Format** command on the NetBeans **Source** menu. The keyboard shortcut for this is: **Alt + Shift + F**.

Lab 2A – Console I/O: Age in Days

In this step-by-step programming exercise we will create a program with console I/O to ask for the user's name and age in years, then return the user's name and age in days. We will use 365.25 days per year as a constant value in the code.

This exercise includes the use of String and numeric data types, variables and constants, simple arithmetic in assignment statements, and console I/O.

The program should:

- get the user's name
- say hello to the user by name and ask for the user's age in years
- calculate the user's age in days
- print the results – the user's age in days.

STEP 1.

Open NetBeans and Start a new NetBeans Java Application project named ageDays.

If any other projects are open in NetBeans, use the *File* menu to close them before continuing. The source code generated by NetBeans will be similar to the source code in Figure 10.

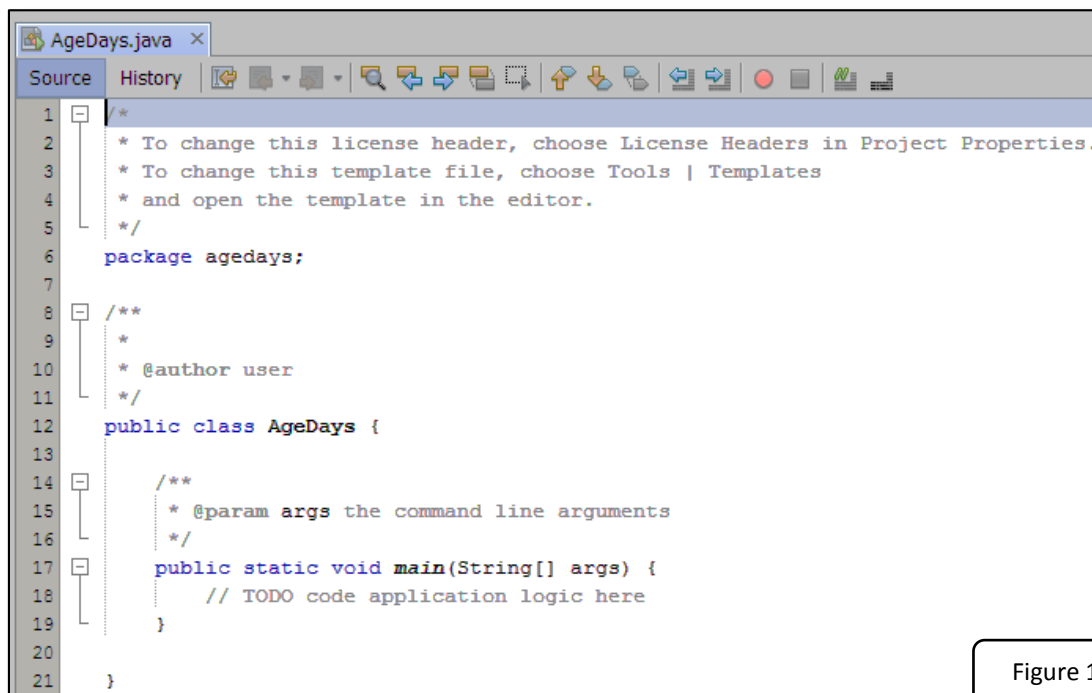


Figure 10

NetBeans Notes – Reformatting java Code

Java code in a NetBeans project can be reformatted to correct indenting by using the **Format** command on the NetBeans **Source** menu. The keyboard shortcut for this is: **Alt + Shift + F**.

STEP 2.

Change the comments in the code as follows:

- Remove the comment on lines 14-16 identifying method parameters
- Remove the comment on lines 8-11 identifying the author
- Change the introductory comment at the start of the source code to include identifying information – the name of the source code file, the nature and purpose of the project, when it was last edited and by whom, as shown in Figure 11.
- Add inline comments labeling the ending braces for the method and the class.

Your code should now resemble that in Figure 11.

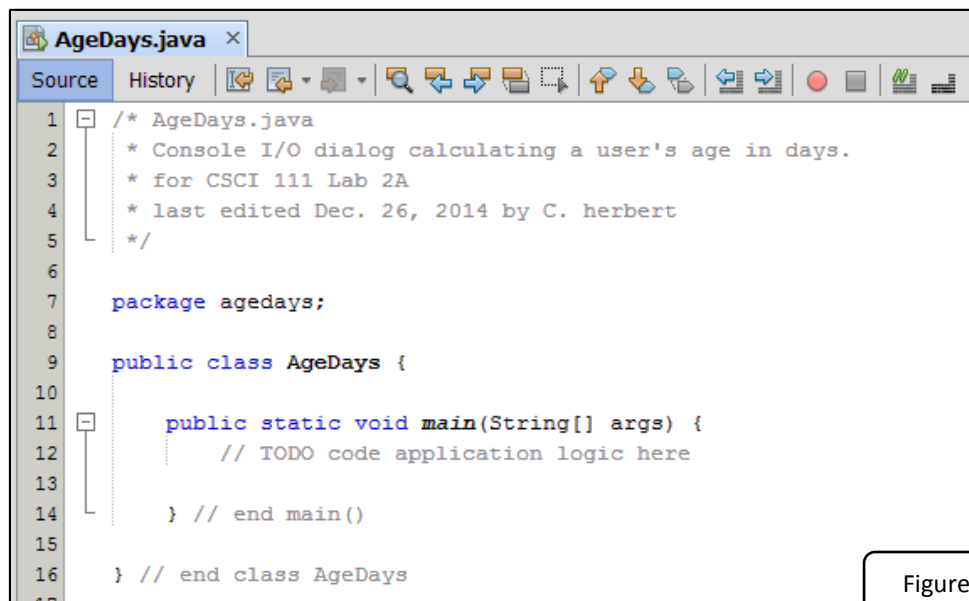


Figure 11

STEP 3.

Add the following comments to the main method in place of the *TODO* comment in the main method.

```
// declare variables
    // user name
    // age (in years)
    // age (in days)
    // constant days per year

// declare an instance of Scanner to read the data stream from the keyboard.

// say hello to the user and ask for the user's name

// say hello to the user by name.

// ask for the user's age in years

// Calculate how many days are in the number of years entered

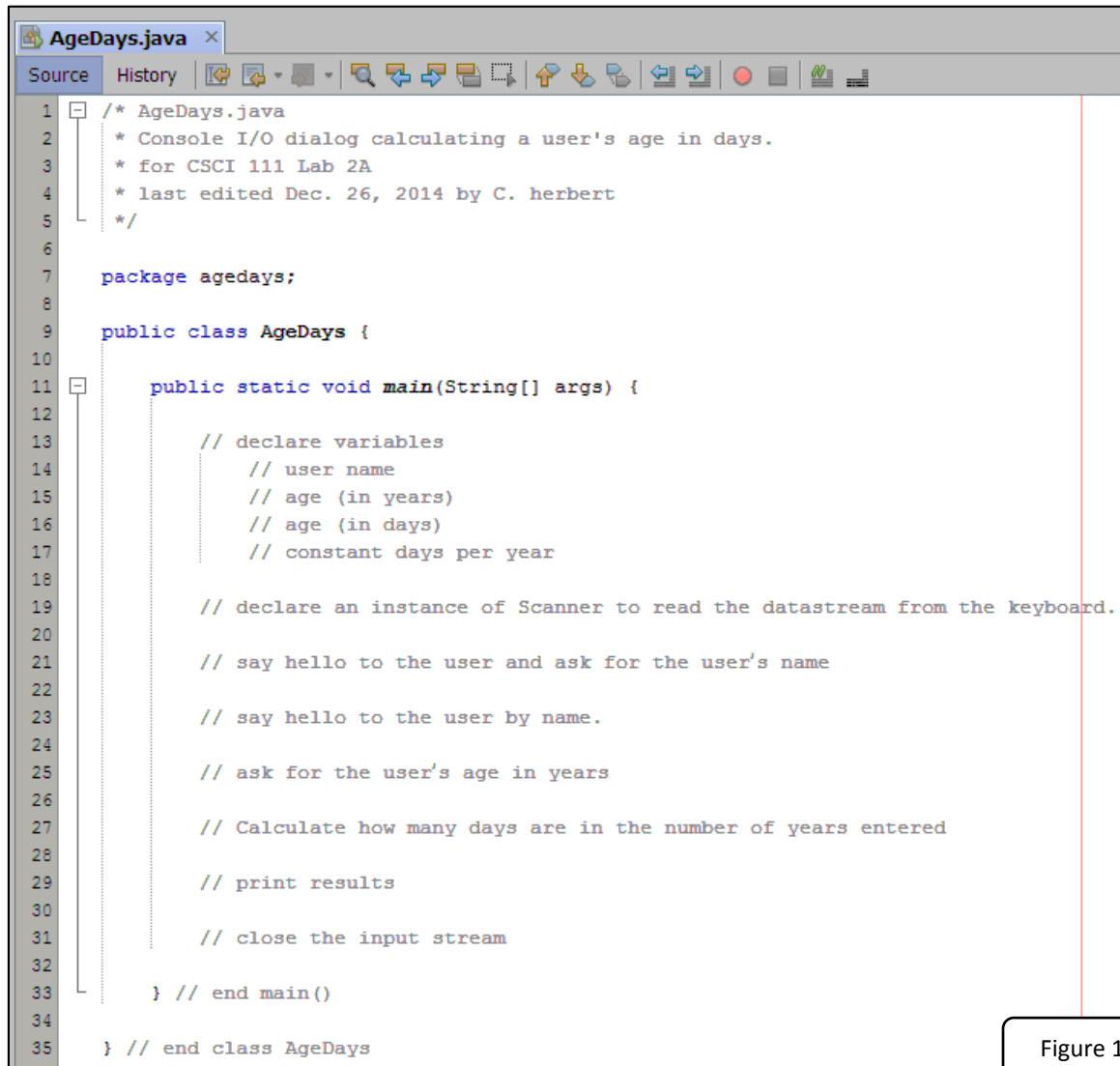
// print results

// close the input stream
```

The code should now resemble that shown in Figure 12. You will need to use the tab key to indent comments properly.

When the comments are in place, we can begin to create code to do what the comments say to do.

s



```
1  /* AgeDays.java
2  * Console I/O dialog calculating a user's age in days.
3  * for CSCI 111 Lab 2A
4  * last edited Dec. 26, 2014 by C. herbert
5  */
6
7  package agetdays;
8
9  public class AgeDays {
10
11     public static void main(String[] args) {
12
13         // declare variables
14         // user name
15         // age (in years)
16         // age (in days)
17         // constant days per year
18
19         // declare an instance of Scanner to read the datastream from the keyboard.
20
21         // say hello to the user and ask for the user's name
22
23         // say hello to the user by name.
24
25         // ask for the user's age in years
26
27         // Calculate how many days are in the number of years entered
28
29         // print results
30
31         // close the input stream
32
33     } // end main()
34
35 } // end class AgeDays
```

Figure 12

STEP 4.

Add declarations to the code as the comments indicate, and remove the *declare variables* comment:

- a String called `name` for the user's name,
- a double called `years` for the age in years,
- a double called `days` for the age in days.
- `final double DAYS_PER_YEAR = 365.25;` to establish the necessary constant.

STEP 5.

Declare an instance of the Scanner class below the appropriate comment:

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

Also, add an import statement right after the package directive near the beginning of the code to make this work:

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

STEP 6.

Add code below the comment `// say hello to the user and ask for the user's name` to do what the comment says:

```
System.out.println("Hello, please enter your name: " );  
name = kb.nextLine();
```

(Can you figure out why we are using the `nextLine()` method instead of just `next()` ?)

STEP 7.

Add code below the comment `// say hello to the user by name to do as the comment says.`

```
System.out.println("Hello, " + name);
```

STEP 8.

Add code below the next comment to ask for the user's age in years.

```
System.out.println("How many years old are you ?");  
years = kb.nextDouble();
```

STEP 9.

Add code in the appropriate location to calculate the user's age in days.

```
days = years * DAYS_PER_YEAR;
```

STEP 10.

Add code to print the result, in a format similar to this: Joe, you are 9865.75 days old.

```
System.out.println(name + ", you are" + days + "days old.");
```

STEP 11.

Add code to close the input stream.

```
kb.close();
```

STEP 12.

Figure 13 on the next page shows the complete program. **Test, run, and debug your code as necessary until it runs correctly.** The *Test Project* and *Run Project* commands are the NetBeans *Run* menu. You should test a program before you run it.

Testing reveals the out could look better. **Modify the code by adding blank `println()` commands to separate the different parts of the output, so that it looks something like this:**

```
Hello, please enter your name:  
Joe
```

```
Hello Joe, please enter your age in years:  
27
```

```
Joe, you are 9865.75 days old.
```

When your program runs correctly, you are finished with this exercise.

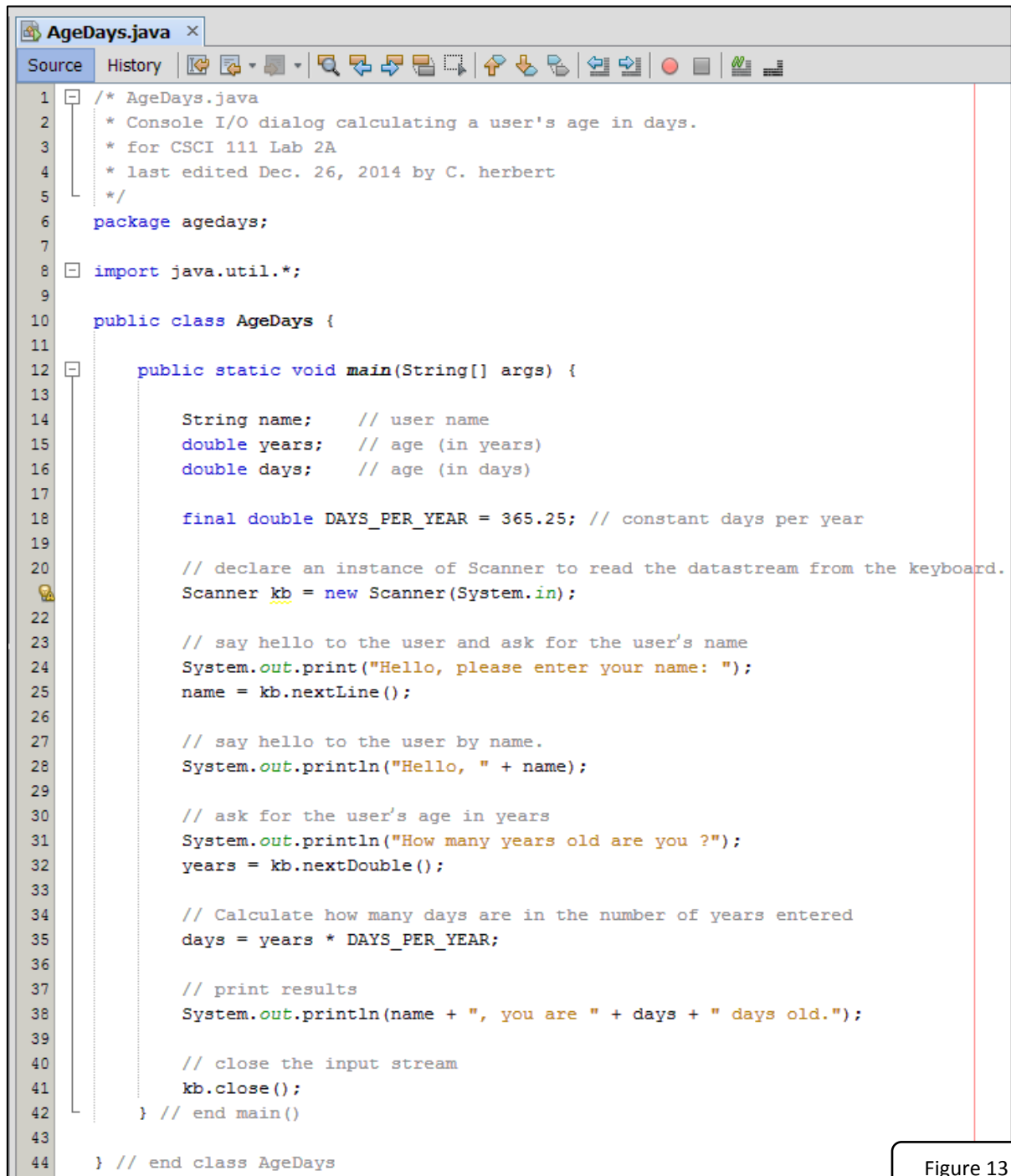


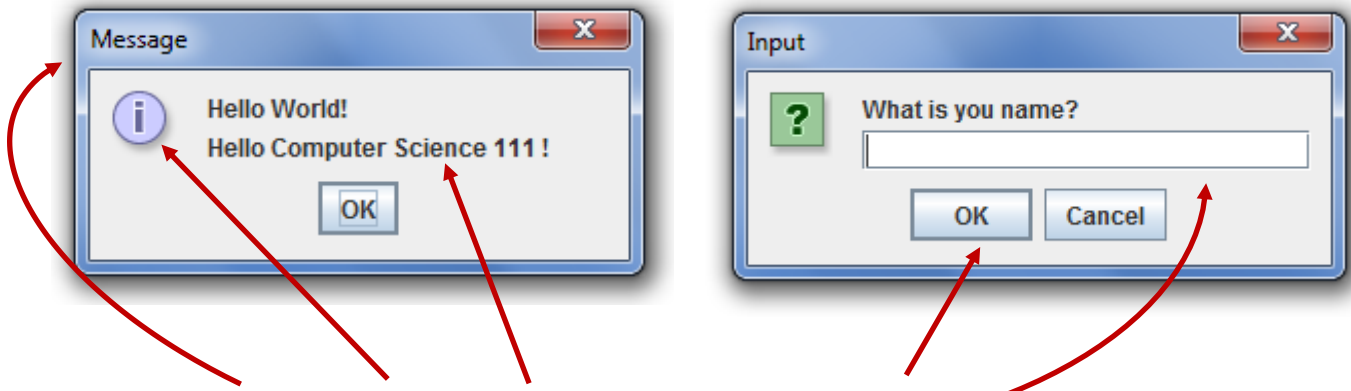
Figure 13

NetBeans Notes

Java code in a NetBeans project can be reformatted to correct indenting by using the **Format** command on the NetBeans **Source** menu. The keyboard shortcut for this is: **Alt + Shift + F**.

Lesson 2.6 JOptionPane Pop-Up Dialog Windows

The Swing API is one of several used for Graphical user Interface (GUI) programming. It includes the **JOptionPane** class, which provides a set of objects for complete pop-up dialog windows, such as the *message dialog window* and the *input dialog window* shown below:



Each window has a **title**, an **icon**, a **text message**, and one or more **buttons**.

A JOptionPane input window also contains a **text entry box**.

The color and style of the pop-up windows depends on settings in each computer's operating system. You will notice that not all of the windows in the examples in this section have the same look and feel.

There are four different types of *JOptionPane* pop-up dialog windows:

- **message dialog window** – simply displays a text message
- **input dialog window** – gets user input as String
- **confirm dialog window** – asks a question with confirmation buttons, such as yes/no/cancel
- **option dialog window** – a programmable dialog window that can be used for various purposes.

In the remainder of this section we will focus on the *message dialog window* and the *input dialog window* for use with simple user I/O. More detailed information about *JOptionPane* pop-up dialog windows can be found on the *JOptionPane Class* documentation page online at:

<http://docs.oracle.com/javase/7/docs/api/javax/swing/JOptionPane.html>

Using the JOptionPane Message Dialog Window





JOptionPane has many methods that can be used to create dialog windows for various circumstances. We will use the `showMessageDialog()` method, which displays a pop-up message dialog window. The specifications for the code are shown by the method header from the *JOptionPane* class:

```
static void showMessageDialog(Component parentComponent, Object message,
                             String title, int messageType, Icon icon)
```

The parameters for this method are:

- **parentComponent** – leave this as *null* for a simple stand-alone pop-up window. The parent component of the window would be named in more complex GUI programming.
- **message** – the message to be displayed in the window. It is most often simply a String message.
- **title** – a String with the title to be displayed in the top of the window. The default is "Message".

- **messageType** – affects the look of the Window, including which icon is displayed. The JOptionPane class has several *messageType* constants that can be used:

messageType constant	Java icon
ERROR_MESSAGE	
INFORMATION_MESSAGE	
WARNING_MESSAGE	
QUESTION_MESSAGE	
PLAIN_MESSAGE	no icon is shown

The default *messageType* is INFORMATION_MESSAGE, The icon for an information message will be shown by default If you do not include a *messageType* constant in the parameters.

Remember, Java is a multi-platform language; some operating systems will override the Java icons and use their own versions of each icon.

- **icon** – This parameter allows a custom icon to be used. We will skip this for now.

The *showMessageDialog()* method may be used as an instruction in a Java program to create a pop-up message window, since it is a void method that does not return a value. The following examples show samples of Java code and the resulting dialog windows.

JOptionPane Import Statement

JOptionPane windows are part of Java's *Swing* package, used for building graphical user interfaces (GUIs). Their use requires an import statement, such as the following:

```
import javax.swing.*;
```

JOptionPane Example 1 – Default Message Dialog Window

In this example, the message is specified and the parameters that follow are not used. The default INFORMATION_MESSAGE icon and title will be shown.

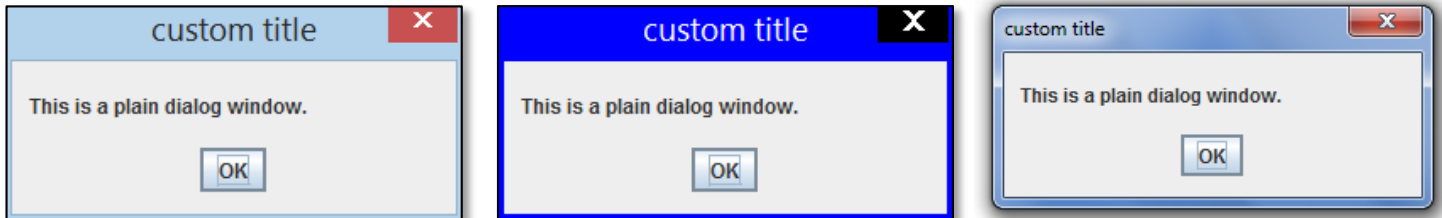
```
// message dialog window with default title and icon
JOptionPane.showMessageDialog(null, "This is a message dialog window.\nHello World!");
```



JOptionPane Example 2 – Plain Message Dialog Window

This example includes the `PLAIN_MESSAGE` parameter with a custom title. No icon will be shown.

```
// plain message dialog window with custom title; no icon is shown in a plain message
JOptionPane.showMessageDialog(null, "This is a plain dialog window.", "custom title",
    JOptionPane.PLAIN_MESSAGE);
```

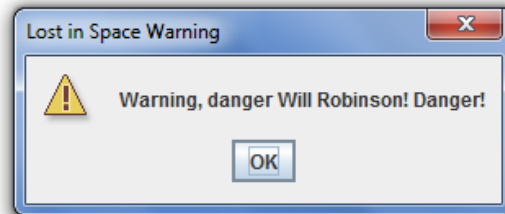


The same code generated all three windows. the look and feel of each was set by the operating system.

JOptionPane Example 3 – Warning Message Dialog Window

This example shows a `WARNING_MESSAGE` with a custom title and warning icon.

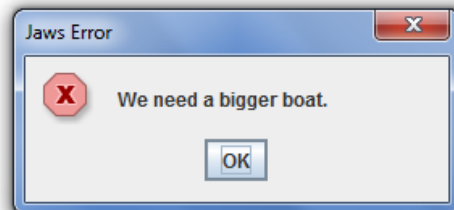
```
// warning message dialog window with custom title and the warning icon
JOptionPane.showMessageDialog (null, "Warning, danger Will Robinson! Danger!",
    "Lost in Space Warning", JOptionPane.WARNING_MESSAGE);
```



JOptionPane Example 4 – Error Message Dialog Window

This example shows an `ERROR_MESSAGE` with a custom title and error icon.

```
// error message dialog window with custom title and the error icon
JOptionPane.showMessageDialog(null, "We need a bigger boat.",
    "Jaws Warning", JOptionPane.ERROR_MESSAGE);
```



Using the JOptionPane Input Dialog Window

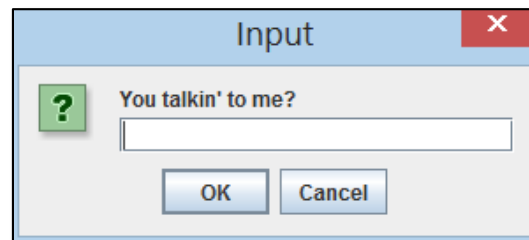
The `JOptionPane.showInputDialog` window will accept user input from the keyboard as a `String`. There are many variations of this method, but the most straightforward is a question-message dialog with one parameter – the question-message to be displayed in the window. Here is the method header:

```
static String JOptionPane.showInputDialog(Object message);
```


The method returns a `String` equal to whatever the user enters in the text entry box in the window. The *message* parameter is a `String`, as is the value returned by the method. We use a `String` variable to capture the input.

JOptionPane Example 5 – Input Dialog Window (String input)

```
// input dialog window getting the user's name. The variable must be a String
// whatever the user enters will be captured as String.
    name = JOptionPane.showInputDialog("What is your name?");
```



This simple, easy to use method works well, but what if we want numeric input? The answer is to use a two-step process:

1. get the user input as a `String`.
2. convert the `String` into a numeric value using a parsing method.

Parsing Numbers from Strings

In linguistics, *to parse a sentence* means to break the sentence down into its component parts of speech. Compilers do something similar with statements in a programming language, by parsing a statement into its component parts. We can also parse `Strings`, which is what we will do here. **To parse a `String`** is to convert the information in a `String`, or part of a `String`, into a value of another data type.

Java has several `Number` classes corresponding to each of the primitive numeric data types. Each one provides a parsing method to parse a `String` into the corresponding numeric data type as follows:

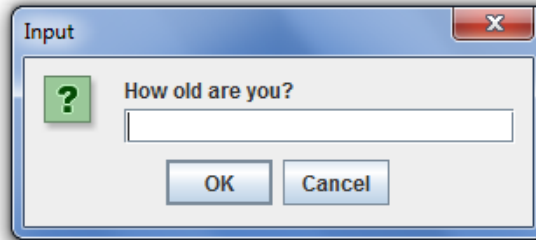
- `static byte parseByte(String)` is part of the `Byte` class
- `static short parseShort(String)` is part of the `Short` class
- `static int parseInt(String)` is part of the `Int` class
- `static long parseLong(String)` is part of the `Long` class
- `static float parseFloat(String)` is part of the `Float` class
- `static double parseDouble(String)` is part of the `Double` class

Since these parsing methods are each static, they are used with the name of the class, such as `Int.parseInt()` or `Double.parseDouble()`. By far, `parseInt()` and `parseDouble()` are the most commonly used parsing methods, as shown in the examples below.

If a parsing method cannot find a valid numeric representation in the `String`, the parsing method will create a *NumberFormatException*, which we learn how to handle later this semester.

JOptionPane Example 6 – Input Dialog Window (double input)

```
// input dialog window getting a double value
// first, get a String
    ageString = JOptionPane.showInputDialog("How old are you?");
```

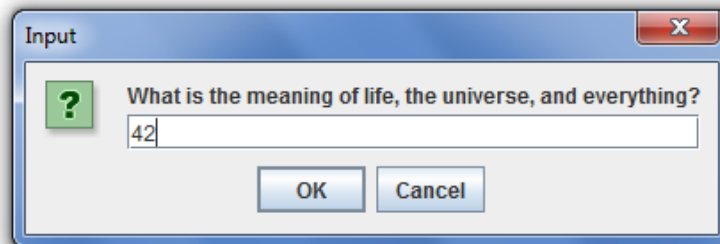


```
//next, parse a double value from the string
    age = Double.parseDouble(ageString);
```

in Example 6, a JOptionPane input dialog window is used to get user input as a String, then a parsing function is used to parse the String into a double value. Example 7 is similar, except that the input String is parsed to an int value.

JOptionPane Example 7 – Input Dialog Window (int input)

```
// input dialog window getting an integer value (image shows user input)
// first, get a String
    answer = JOptionPane.showInputDialog("What is the meaning of life,
        the universe, and everything?");
```

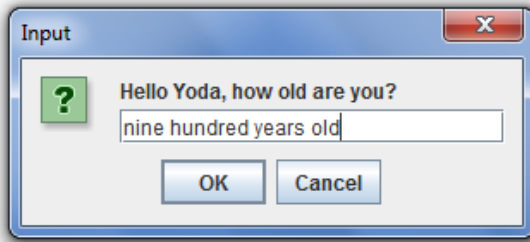


```
//next, parse a double value from the string
    adamsNumber = Int.parseInt(answer);
```

Input dialog windows are convenient, but they do not take the place of a more complete graphical user interface, which we will study later in a later chapter.

Exceptions When Parsing Numbers from Strings

What happens if we try to parse a number from a String that does not have a numeric value? In NetBeans, the exception that is created will cause the program to fail, and we will see an error:



```
Exception in thread "main" java.lang.NumberFormatException: For input string: "nine hundred
years old"
    at sun.misc.FloatingDecimal.readJavaFormatString(FloatingDecimal.java:1241)
    at java.lang.Double.parseDouble(Double.java:540)
    at joptionpanedemo.JOptionPaneDemo.main(JOptionPaneDemo.java:36)
Java Result: 1
```

Figure 14
Exception warning

This message tells us a *NumberFormatException* was created by the input String "*nine hundred years old*" in the *readJavaFormatString* method, called by the *parseDouble* method, called by the main method in our *JOptionPaneDemo* program. The first line tells us the kind of exception created. The last line tells us the ultimate cause of the error – line 36 in the method *JOptionPaneDemo.main*. Line 36 is not wrong, but that is the line which was being executed when the error occurred. This is line 36:

```
age = Double.parseDouble(inputString);
```

The computer could not parse *inputString* "*nine hundred years old*", because it has no numbers to parse.

Exceptions are objects, so before learning to handle exceptions in java code, you need to learn more about objects. These topics will be covered later in this course and in Computer Science 112.

Checkpoint 2.6

1. What does the *JOptionPane* class provide?
2. Describe the four different types of *JOptionPane* pop-up dialog windows.
3. What determines the look and feel of *JOptionPane* pop-up dialog windows?
4. What datatype does a *JOptionPane* input dialog window return?
5. Describe how to parse text from *JOptionPane* input dialog windows into numeric data types.

NetBeans Notes – Test and Run Source Code

It is a good idea to test compiled source code before running a NetBeans project. The **Test** option on the **Run** menu will compile the code and report any compile time errors before attempting to run the code. The **Run** option on the **Run** menu will run the project, or compile *and* run if the code has been changed since it was last compiled.

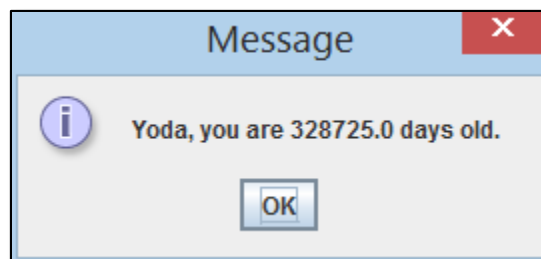
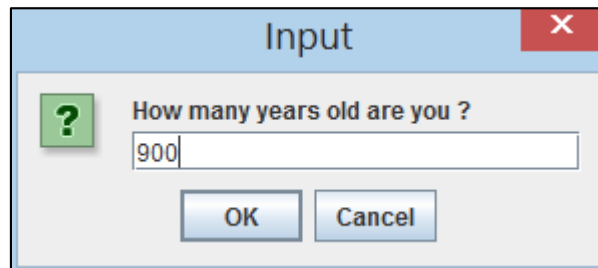
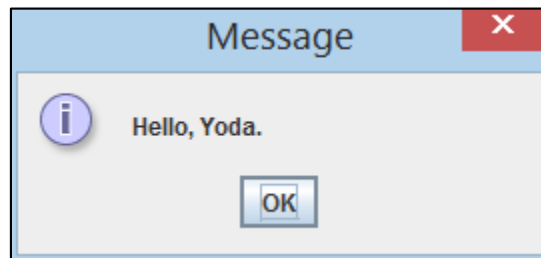
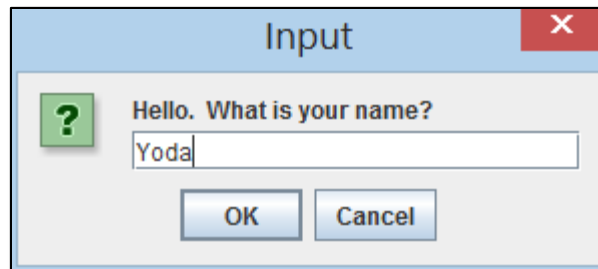
Both test and run will save all project files.

Test Project and *Run Project* also have keyboard shortcuts: Test is **Alt+F6**, Run is **F6**.

Lab 2B – Pop Up Windows Dialog: Age in Days

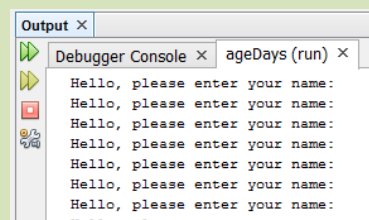
In this exercise we will modify the code from the *ageDays* program in Lab 2A to use JOptionPane windows for input and output in place of console I/O.

Here is a sample of the I/O from the resulting program.



NetBeans Notes – Stopping a Running Program

A NetBeans project's *Output* window has icons that can be used to stop and re-start a NetBeans program. The rectangular red stop button is often used for a “runaway program” that will not stop on its own or for a GUI that will not close properly.



The new code will have four JOptionPane windows: two message windows, and two dialog windows. The code for the calculation will be in our program between the code segments for the last two windows. The windows shown above resulted from running the code on a Windows 8.1 system. The operating system set the look and feel of the JOptionPane windows.

STEP 1.

Download and unzip the *ageDaysComplete* NetBeans project included with the files for this chapter, then open the project in NetBeans. It contains the completed ageDays project from Lab 2A. You should unzip the project folder in a place that will be easy to access from within NetBeans, such as the desktop.

The source code for the project is shown in figure xx on page 35. The comment at the top of the code refers to the original code. **Change the comment as follows:**

1. delete the word starting on line 3
2. update the date to the current date
3. update the most recent editor's name to be your name

The existing import statement on line 8 is needed for console I/O, but not for JOptionPane I/O. Instead we need an important statement that will enable the use of Swing objects.

STEP 2.

Replace the existing import statement – *import java.util.*;* – with an import statement to enable the use of Swing objects:

```
import javax.swing.*;
```

STEP 3.

There is no need to declare a Scanner class object in this program.

Delete the code in the main method that declares a Scanner class object and the comment above it:

```
// declare an instance of Scanner to read the datastream from the keyboard.  
Scanner kb = new Scanner(System.in);
```

You should also delete any extra blank lines so the code now looks something like this:

```
public static void main(String[] args) {  
  
    String name;    // user name  
    double years;   // age (in years)  
    double days;    // age (in days)  
  
    final double DAYS_PER_YEAR = 365.25; // constant days per year  
  
    // say hello to the user and ask for the user's name  
    System.out.print("Hello, please enter your name: ");  
    name = kb.nextLine();  
  
    // say hello to the user by name.  
    System.out.println("Hello, " + name);  
  
    // ask for the user's age in years  
    System.out.println("How many years old are you ?");  
    years = kb.nextDouble();  
}
```

Figure 15

Several code segments in the method perform console I/O. Now we need to replace these with code to do the same things using JOptionPane input dialog windows.

STEP 4.

The first console I/O code segment to be replaced is just below the constant declaration where we find a comment and two lines forming a prompt and capture pair to ask for the user's name.

Replace the two statements for the first prompt and capture pair in the source code with a statement to do the same thing with a JOptionPane input dialog window:

```
name = JOptionPane.showInputDialog("Hello. What is your name?");
```

This part of the code should now look like this:

```
final double DAYS_PER_YEAR = 365.25; // constant days per year

// say hello to the user and ask for the user's name
name = JOptionPane.showInputDialog("Hello. What is your name?");

// say hello to the user by name.
System.out.println("Hello, " + name);

// ask for the user's age in years
System.out.println("How many years old are you ?");
years = kb.nextDouble();
```

Figure 16

STEP 5.

Replace the *println()* statement that says hello to the user by name with code for a JOptionPane message dialog window to do the same:

```
JOptionPane.showMessageDialog(null, "Hello, " + name + ".");
```

STEP 6.

Replace the prompt and capture pair of statements with a statement to do the same thing using a JOptionPane input dialog window:

```
ageString = JOptionPane.showInputDialog("How many years old are you ?");
```

Notice that the variable is named *ageString*. We have not yet declared this variable. The input from a JOptionPane input dialog window is always a String. We need to declare this variable near the top of the method so that we can use it. We will also need to add code after the code for the JOptionPane input dialog window to parse the String value *ageString* to the double value *years*.

STEP 7.

Add a statement to the code near the top of the method to declare `ageString` as a `String` variable to be used in the method, so that the variable declarations near the beginning of the method look like this:

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

    String name;           // user name
    String ageString;      // age as a String from user input
    double years;          // age (in years)
    double days;           // age (in days)

    final double DAYS_PER_YEAR = 365.25; // constant days per year
```

Figure 17

STEP 8.

Add a statement after the code for the `JOptionPane` input dialog window to parse the `String` value `ageString` to the double value `age`.

```
years = Double.parseDouble(ageString);
```

The input and the calculate segments of the code should now look like this:

```
// say hello to the user and ask for the user's name
name = JOptionPane.showInputDialog("Hello. What is your name?");

// say hello to the user by name.
JOptionPane.showMessageDialog(null, "Hello, " + name + ".");

// ask for the user's age in years
ageString = JOptionPane.showInputDialog("How many years old are you ?");
years = Double.parseDouble(ageString);

// Calculate how many days are in the number of years entered
days = years * DAYS_PER_YEAR;
```

Figure 18

Next, we need to convert the output from console I/O to `JOptionPane` I/O. We can also remove the close command leftover from the scanner class, since it is no longer needed.

STEP 9.

Replace the `println()` statement showing the result of the calculation with code for a `JOptionPane` message dialog window to do the same:

```
JOptionPane.showMessageDialog(null, name + ", you are " + days + " days old.");
```

STEP 10.

Delete the close statement and the comment that precedes it. Adjust the line spacing as needed.

```
// close the input stream
kb.close();
```

STEP 11.

Your program is now complete.

Test and debug the project as necessary before closing the project and exiting NetBeans.

Key Terms in Chapter 2

After completing this chapter, You should be able to define each of the following key terms:

ad hoc polymorphism, 18	functions, 17	option dialog window, 38
alphanumeric character, 8	hardcoding, 15	order of evaluation, 23
ASCII, 7	import statement, 29	output data streams, 27
assignment statement, 17	increment, 19	parametric polymorphism, 18
associative, 24	Input data streams, 27	parse, 41
boolean, 6	input dialog window, 38	polymorphism, 17
built-in data types, 5	int (data type), 10	postfix operation, 20
byte (data type), 10	Java identifier, 16	prefix operation, 20
byte addressable, 7	java.lang package, 20	primitive data type, 5
cast operator, 26	JOptionPane, 38	prompt and capture, 28
char, 7	Keywords, 16	qualified name, 16
character, 7	literal, 6	raw data stream, 27
commutative, 24	local variables, 14	reference data type, 5
compound assignment, 20	long (data type), 10	remainder operation, 19
concatenation, 18	mantissa-exponent format, 10	Scanner class, 28
confirm dialog window, 38	Math class, 21	scope of a variable, 14
constant, 6	message dialog window, 38	short (data type), 10
data type, 5	modulus operation, 19	significand, 10
declare a variable, 14	narrowing primitive conversion, 25	simple name, 16
decrement, 20	negation, 19	String literal, 7
delimiter, 27	non-numeric characters, 8	String, 7
documentation first programming, 30	null variable, 14	token, 27
double (data type), 12	operands, 17	tokenized data stream, 27
final, 15	operator overloading, 18	type casting, 25
float (data type), 12	operator precedence, 23	unary operation, 19
front loading declarations, 15	operators, 17	Unicode, 7
function overloading, 18		UTF-16, 7
		variable, 6
		whitespace, 27

NetBeans Notes – Backing Up Project Files

The project folder for each NetBeans project contains the files needed for the project. It a good idea from time to time to make a backup copy of the project folder – especially for any important projects and for any projects recently edited. Two copies of any important NetBeans project should be stored in different locations on different devices to prevent a loss of the project in case of a systems failure.

Chapter 2 - Questions

1. Why do we have different data types? How can a programmer determine which data type to use for a variable? What is the range of values for a data type related to? What is the difference between a primitive data type and a reference data type?
2. What data types are built into Java in each of the following categories: Boolean data, numeric data, and text data?
3. What is the minimum storage space needed for a boolean value? Why is a boolean variable in Java stored as a byte rather than a bit?
4. How are integers entered in Java? How are they stored? What special data type does Java have for unsigned integers? What is the only difference between any two of the four integer data types in Java? Which of the Java integer data types is used most often?
5. What formats does Java use to store floating point numbers? What is the accuracy and range of values for Java's two floating point formats? Which of the two Java floating point data types is used most often?
6. Where can we find out more about data types in Java?
7. What character set is used for char and String data in Java? Where can String literals be used in Java? How do we indicate String literals in Java?
8. How is the scope of a local variable determined in Java?
9. How is a constant different from a variable in Java? How are constants declared in Java? When does Java replace constants in code with their values? Why is hardcoding rather than using constants a bad habit to develop?
10. What rules determine how Java identifiers are named? What is camel case? How should camel case be used in naming variables, methods and classes in Java? What is the difference between a simple name and a qualified name in Java? What characters may be used in the names of Java variables?
11. What do we find to the left of the equals sign in a Java assignment statement? What do we find on the right? What will happen if an expression yields a value that is not of the same data type as the variable to which it is being assigned? Why is this a bad idea?
12. What is the difference between an operator and an operand in an expression in Java? What actually determines which operation will be performed in Java expressions? What is this a form of? What operation will be performed on String operands with the plus sign (+) as an operator?
13. What is a *narrowing primitive conversion*, and why is it such a problem in computer programming?
14. What are the eight arithmetic operations that may be performed on numeric data in Java? Which of these are unary operations, and what does that mean?
15. What is the difference between postfix and prefix increment operations? Illustrate your answer with examples that show the difference.
16. What are some of the methods in the java.lang.Math class, and how are they used in Java programs? How do some of these methods exhibit polymorphism? Where can we find out more about the Java Math class?

17. How is the order of operations determined in a programming language? What is the order of precedence for arithmetic operations in Java? How should fractions with operations in the numerator and the denominator, such as $\frac{x+3}{x-3}$, be entered in Java as numeric expressions to preserve the intended order of operations?
18. Which arithmetic operations in Java are commutative and associative? Which two-operand arithmetic operations are not commutative and associative?
19. How are Java I/O data streams usually organized? What do they use as delimiters? What are some of the Scanner class methods we can use to read data from an I/O stream? What must we do to use them in Java?
20. What is the difference between a message dialog window and an input dialog window? What are the five message type constants used for JOptionPane message dialog windows and what are the icons that go with each of them?

Chapter 2 – Exercises

1. Identify appropriate data types for each of the following situations with a reason for your choice:
 - a. hours, rate, and gross pay in a payroll program.
 - b. the number of people in each census district; the average number of people in each census district.
 - c. the number of people on a 40-passenger SEPTA bus. The code will run on a small processor embedded in the bus's electronic passenger counter.
 - d. Serial numbers in a program to keep track of digital recordings.
 - e. each of the following fields in a student data base program – first name, last name, student number, city, state, zip code, major, GPA, currently enrolled?
2. In the following list of variable names, describe which are invalid in Java and why, which are valid but violate programming conventions and why, and which are valid.
x, sum, sum1, float, name\$, name_42, native, joeSmith, 1root, BattingAverage, status?, GPA

NetBeans Notes – Help Menu

The **Help Contents** section of the NetBeans **Help** menu has links to online tutorials about using NetBeans to develop software in various languages and links to chapters in the **Developing Applications with NetBeans IDE User's Guide**.

3. Show the output for each of the following:

- a. [Note: all variables are double – the square root of 10 is 3.162278]

```
num = 10.0;
square = num*num;
sroot = Math.sqrt(num);
System.out.println("number\tsquare\square root");
System.out.println(num + "\t" + square + "\t" + sroot);
```

- b.
- ```
System.out.print("This assignment ");
System.out.print("is due on ");
System.out.print("September, ");
System.out.println("8th.");
```

- c.
- ```
System.out.print("This assignment ");
System.out.print("is due on");
System.out.println("September, ");
System.out.println("8th.");
```

- d.
- ```
System.out.print("This assignment ");
System.out.print("is due on\n");
System.out.print("\tSeptember, ");
System.out.println("8th.");
```

## 4. Rewrite each of the following formulas as Java assignment statements:

a.  $C = \frac{5}{9}(F - 32)$

Fahrenheit to Celsius conversion

b.  $\text{payment} = \frac{ra}{1 - (1+r)^{-n}}$   
periods)

loan payment (a = loan amount, r = rate, n =

c.  $\text{root} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

calculating two roots of a quadratic equation

d.  $\text{dist} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

distance from  $(x_1, y_1)$  to  $(x_2, y_2)$ 

e.  $\text{haversine} = \frac{1 - \cos(\theta)}{2}$

haversine function used in maritime navigation

## 5. The yards and feet conversion program in section 2.3 is incomplete. Finish the program and get it to run in NetBeans. Make sure it is properly documented.

### I-P-O software

A common simple design pattern in programming is input-processing-output (I-P-O).

In problems 6 through 10, create interactive I-P-O software that asks the user for some input, processes the data, then outputs the result.

You should use variable names that are more meaningful than those in exercise 3, above, or in the formulas below. Use constants where appropriate. Make your output look attractive, useful to the user, and easy to understand. Keep your code readable, easy to understand, and well-organized.

#### 6. World City Temperature Celsius to Fahrenheit Converter

Input:

- ask for a city name
- ask for the current temperature in the city temperature in degrees Celsius, using the name of the city

Processing:

- convert to degrees Fahrenheit using the formula:

$$F = \left( \frac{9}{5} C \right) + 32$$

Output:

- a statement of the form:  
*The current temperature in London is 20 °C, which is 68 °F*  
[ Note: the degree symbol is Unicode \u00b0 ]

#### 7. Monthly Loan Payment Calculator

Input:

- the address of the property
- the amount of the loan
- annual interest rate, (Entered as a decimal. For example, 4.5% is .045)
- number of monthly payments

Processing:

- calculate the effective monthly interest rate by dividing the annual rate by 12.0
- calculate the monthly payment using the formula in exercise 4b, above

Output:

- the amount of the loan
- the annual interest rate
- the number of monthly payments
- the amount of each monthly payment

[Note: test data – \$100,000 at 5% for 30 years is a payment of \$536.82]

## 8. Change for a dollar.

Input:

- using short integers, ask the user for a number of cents less than 1 dollar

Processing:

- calculate the number of quarters, dimes, nickels and pennies in the amount.  
We do this using the division and remainder operations. Think about how you would do it, then design a program to do the same. How many quarters? How much is left over? How many dimes in that amount, and so on?

Output:

- a neatly organized statement of the form:

87 cents is:

3 quarters

1 dime

0 nickels

2 pennies

## 9. Area, Volume, and Surface Area

Input:

- ask the user to input a distance in inches

Processing:

- calculate the area of:
  - a circle with that radius  $\text{area} = \pi r^2$
  - a square with that side  $\text{area} = s^2$
- calculate the volume of:
  - a sphere with that radius  $\text{volume} = \frac{4}{3} \pi r^3$
  - a cube with that side  $\text{volume} = s^3$
- calculate the surface area of
  - a sphere with that radius  $\text{surface area} = 4 \pi r^2$
  - a cube with that side  $\text{surface area} = 6 s^2$

Output:

- an attractive and neatly organized display of the results.

## 10. Identify five errors in the code on the next page that will stop the program from compiling or running:

```
package futureinvestment;
import java.util.Scanner;

public class FutureInvstment
{
 public static void main(String[] args) {
 {
 double futureInvestment; // the future investment
 double investmentAmount; // the initial investment
 double annualInterestRate; // the interest rate multiplied by 100
 double monthlyInterestRate // the monthly interest rate
 double numberOfYears; // the number of years of investment

 Scanner input = new Scanner(System.in);

 //get amount
 System.out.print("Enter the investment amount: ");
 investmentAmount = input.nextDouble();

 //get annual rate and convert to monthly rate
 System.out.print("Enter the annual rate: ");
 annualInterestRate = keyboard.nextDouble();

 //get number of years
 System.out.print("Enter the number of years: ");
 numberOfYears = input.nextDouble();

 //PROCESSING
 annualInterestRate / 12 = monthlyInterestRate;
 futureInvestment = investmentAmount *Math.pow((1 + monthlyInterestRate / 100),numberOfYears * 12);

 System.out.print("Accumulated value is ");
 System.out.println(futureInvestment);
 } //end main()
 } // end class FutureInvstment
```

— End of Chapter 2 —