Chapter III

Java Simple Data Types

Chapter III Topics

3.1 Introduction
3.2 Declaring Variables
3.3 The Integer Data Types
3.4 The Real Number Data Types
3.5 Numerical Representation Limits
3.6 Arithmetic Shortcut Notations
3.7 The char and String Data Types
3.8 The boolean Data Type
3.9 Declaring Constants
3.10 Documenting Your Programs
3.11 Mathematical Precedence
3.12 Type Casting
3.13 Summary

3.1 Introduction

In the early days of your math courses only constants were used. You know what they are. Constant numbers are **5**, **13** and **127**. You added, subtracted, multiplied and divided with numbers. Later, you had more fun with fractions and decimal numbers. At some point variables were introduced. In science and mathematics it is useful to express formulas and certain relationships with variables that explain some general principle. If you drive at an average rate of **60 mph** and you continue for **5 hours** at that rate, you will cover **300 miles**. On the other hand, if you drive at a rate of **45mph** for **4 hours**, you will travel **180 miles**. These two problems are examples that only use constants. The method used for computing this type of problem can be expressed in a general formula that states:

Distance = Rate × Time

The formula is normally used in its abbreviated form, which is **d = r × t**. In this formula **d**, **r** and **t** are variables. The meaning is literal. A variable is a value that is able to change. A constant like **5** will always be **5**, but **d** is a variable, which changes with the values of **r** and **t**. Both **r** and **t** are also variables.

Variables make mathematics, science and computer science possible. Without variables you are very limited in the type of programs that you can write. In this chapter you will learn how to use variables in your programs.

3.2 Declaring Variables

A computer program is made up of words, which usually are called *keywords*. The keywords in a program have a very specific purpose, and only keywords are accepted by the compiler. A compiler will only create a bytecode file if the source code obeys all the Java syntax rules. The first rule is that only keywords known to the Java compiler can be used in a program. Another syntax rule is that each variable needs to specify its data type.

How about first concentrating on one rule at a time? So what exactly is a keyword? You have seen certain words like **void**, **static** and **println**, but that only tells part of the story. Keywords can be divided into three categories.

Java Keywords

- **Reserved Words**
- **Pre-defined Java Identifiers**
- **User-defined Identifiers**

Reserved Words

Reserved words are part of the Java language the same way that **table**, **walk** and **mother** are part of the English language. Each reserved word has a special meaning to Java and these reserved words cannot be used as an identifier, or variable name, for any other purpose in a program. Reserved words that you have seen so far are **public, void** and **static**.

Predefined Identifiers

Java has a large number of libraries that enhance the basic Java language. These libraries contain special program modules, called *methods* that perform a variety of tasks to simplify the life of a programmer. You have seen two methods so far: **print** and **println**. They are special routines that display output in a text window. Both **print** and **println** are called *predefined identifiers*.

User-Defined Identifiers

The third and last type of word used in a program is selected by the programmer. Programmers need to select an identifier for each variable that is used in a program. Variables are used in a program for many purposes, which you will see shortly. You already have familiarity with the general concept of a variable from mathematics. It was stated earlier that we say **distance = 60 * 10** to compute the distance traveled at 60 mph for a 10 hour period. That statement comes from

the general formula of **d = r x t**, which uses three variables. Make sure your identifier selection is neither a **reserved word** nor a **predefined identifier**. The rules for naming an identifier are simple. The identifier can use alphanumeric characters and the underscore character. Additionally, you need to be sure that the identifier starts with an alpha character. You will note that this rule is identical to the rule for naming the **class** identifier of your program. Any identifier created by the programmer is called a *user-defined identifier*.

Fine, you have accepted the need to declare the variables that are used in a program. You have sympathy with the compiler who needs to sort out the proper keywords from the typos, mistakes, and general attempts made by - sometimes clueless - programmers. Of course, you do not fall in the clueless category. Now what about this second syntax rule mentioned earlier, something about indicating a data type with a variable? What is that all about?

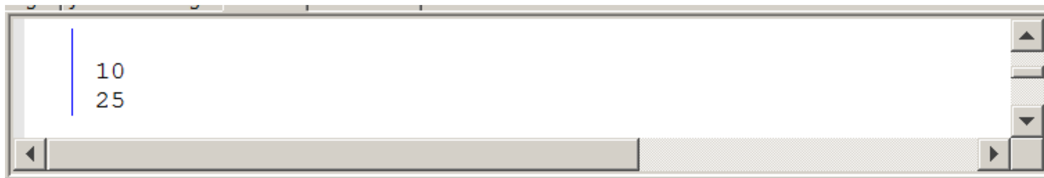
The data type rule is for the purpose of using memory efficiently. All variable values need to be stored in memory during program execution. As long as the program is alive and the variable is in use, its value will be stored somewhere in the computer's memory. It is certainly possible to skip the whole data type scene and give the same exact memory to each variable. Now is that not the same as stating that every room in a building needs to be the same size? How about meeting rooms, closets, offices, bathrooms and dining halls; should they all be the same size? No, that is too silly; a room size is designed for its purpose. Building materials are expensive and lease rates are outrageous. A thrifty business person makes sure to rent the proper amount of space; no more and no less.

Variables are needed to store information such as a single character, which can be placed in one byte or two bytes of memory. Other variables store large numbers that require four or as many as eight bytes of memory. There are also variables that need to store the street address of a customer. Such values may require around 50 bytes of memory. The efficient and practical approach is to declare all variables before they are used and let the compiler know how much memory is required. Once the compiler sees the data type of the variable, memory space will be reserved or allocated for the specified data type.

There is a good selection of simple data types in Java, but for starters take a look at program **Java0301.java**. This program uses an integer data type, which is abbreviated **int** in Java. In figure 3.1 you see that the data type, **int**, starts the declaration statement, followed by the variable identifier, which in this case is either **a** or **b**. This program also introduces the *assignment statement*, which is a statement that assigns a value to a variable. The equal sign is the assignment operator, and does not create an equation. Novice programmers often think that a program statement, like **a = 10;** is an equation. Such a statement should be read as *a becomes 10* or *10 is assigned to a*, but not *a equals 10*.

Figure 3.1

```
// Java0301.java
// This program demonstrates how to declare integer variables with <int>,
// and it shows how to display the value of a variable with <println>.

public class Java0301
{
	public static void main (String[] args)
	{
		int a;
		int b;
		a = 10;
		b = 25;
		System.out.println();
		System.out.println(a);
		System.out.println(b);
		System.out.println();
	}
}
```

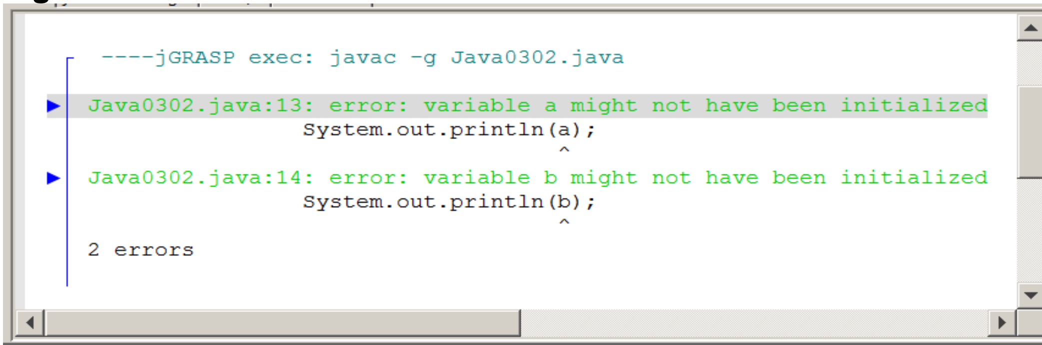


Java is picky about a variety of things. Experienced programmers like a program language to be relaxed so as to give them lots of slack. Novice programmers benefit more from a tight leash that allows little breathing space. Most people agree that Java does not let you jump around much. As a matter of fact, Java insists that a variable is assigned a value before the variable value is used.

Figure 3.2

```
// Java0302.java
// This program is Java0301.java without assigning values to the variables.  Java does
// not compile a program that attempts to use unassigned "simple" data types.

public class Java0302
{
	public static void main (String[] args)
	{
		int a;
		int b;
		System.out.println(a);
		System.out.println(b);
	}
}
```

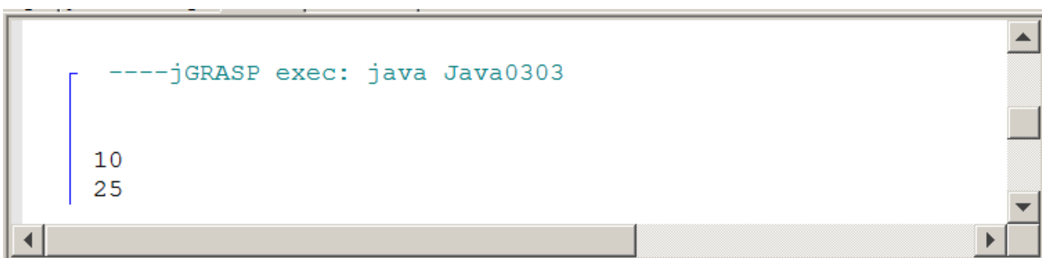
Figure 3.2 Continued



Program **Java0302.java** is almost identical to the previous program minus the assignment statements. This makes Java very unhappy and you are rewarded with some error messages. It is a good habit to assign an initial value to a variable as soon as the variable is declared. It takes less program code to use such an approach and you remember to take care of the variable the same time that you first introduce the variable to your compiler. It is possible to combine the declaration statement and the assignment statement into one program statement. This is shown in figure 3.3 by program **Java0303.java**, and you will note that it produces the exact same output as the earlier program shown in figure 3.1.

Figure 3.3

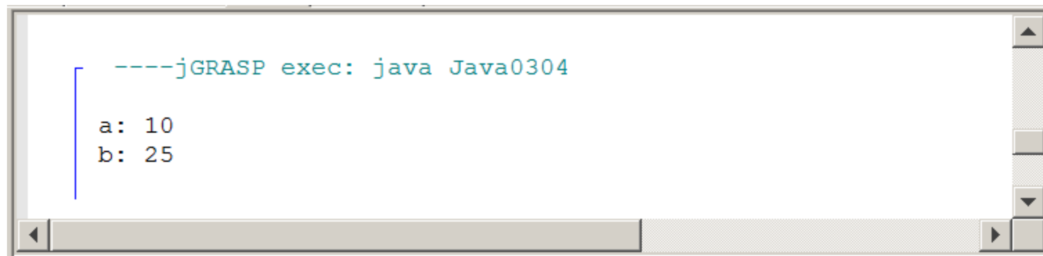
```
// Java0303.java
// This program demonstrates that it is possible to declare a variable
// identifier and initialize the variable in the same statement.
// It is a good habit to initialize variables where they are declared.

public class Java0303
{
	public static void main (String[] args)
	{
		int a = 10;
		int b = 25;
		System.out.println();
		System.out.println(a);
		System.out.println(b);
		System.out.println();
	}
}
```



The early program examples in the last chapter displayed *string literals*, which were contained between the quotes of a **println** statement. Now you see that the double quotes are gone, and the value of the variable is displayed by **println**. You are probably impressed by the Java **println** method, but wait there is more. You can combine the literal character string output with the variable value output by using the plus operator, as is shown by program **Java0304.java**, in figure 3.4.

Figure 3.4

```
// Java0304.java
// This program combines output of literals and variables.
// "a: " is a string literal, which displays the characters a:
// a is an integer variable, which displays its integer value 10.

public class Java0304
{
	public static void main (String[] args)
	{
		int a = 10;
		int b = 25;
		System.out.println("a: " + a);
		System.out.println("b: " + b);
	}
}
```



3.3 The int Data Types

The previous section introduced the notion of declaring variables. You will see many more program examples with variable declarations. In an attempt to be organized, the additional program examples will be shown in a section for each major data type. You did already see some examples with the **int** data type, but as you will see there is quite a bit more to be said about integers. You also need to know how to perform arithmetic operations with integers.

Java supports four integer data types. The table, shown in figure 3.5, indicates four integer types. The programs in Exposure Java and questions on the AP Computer Science Examination only use the **int** type. The other data types allow efficient use of memory if programs require smaller or larger integer values.

Figure 3.5

Java Integer Data Types		
Data Type	**Bytes**	**Minimum & Maximum Values**
byte	1	-128 . . . 127
short	2	-32,768 . . . 32,767
int	4	-2,147,483,648 . . . 2,147,483,647
long	8	-9,223,372,036,854,775,808 . . . 9,223,372,036,854,775,807

Integer data types in Java have five arithmetic operations. You may have expected the four basic operations of addition, subtraction, multiplication and division, but Java adds *modulus*, or *remainder*, division to the list. Program **Java0305.java** in figure 3.6, demonstrates each of the operations.

Figure 3.6

```
// Java0305.java
// This program demonstrates the five integer operations.

public class Java0305
{
	public static void main (String[] args)
	{
		int a = 0;
		int b = 25;
		int c = 10;
		a = b + c;                                        // Addition
		System.out.println(b + " + " + c + " = " + a);
		a = b - c;                                        // Subtraction
		System.out.println(b + " - " + c + " = " + a);
		a = b * c;                                        // Multiplication
		System.out.println(b + " * " + c + " = " + a);
		a = b / c;                                        // Integer Division
		System.out.println(b + " / " + c + " = " + a);
		a = b % c;                                        // Remainder Division
		System.out.println(b + " % " + c + " = " + a);
	}
}
```

Figure 3.6 Continued

```
----jGRASP exec: java Java0305

25 + 10 = 35
25 - 10 = 15
25 * 10 = 250
25 / 10 = 2
25 % 10 = 5
```

There is little explanation needed for addition, subtraction and multiplication. Your biggest concern is that you need to remember to use an **asterisk *** for multiplication. Division can be a little confusing. Java recognizes two types of division with the **int** type: *integer quotient division* and *integer remainder division (modulus division)*. Look at the examples in figure 3.7 to understand the difference.

Figure 3.7

Integer Quotient Division Examples

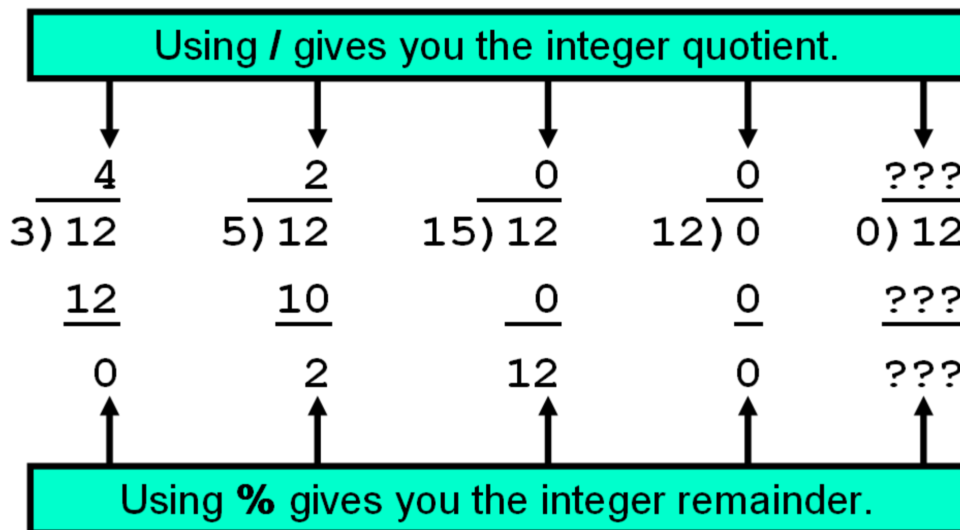
```
12  /  1   =  12          12  /  8   =  1
12  /  2   =  6           12  /  9   =  1
12  /  3   =  4           12  /  10  =  1
12  /  4   =  3           12  /  11  =  1
12  /  5   =  2           12  /  12  =  1
12  /  6   =  2           12  /  13  =  0
12  /  7   =  1           12  /  0   =  undefined
```

Integer Remainder Division Examples

```
12  %  1   =  0           12  %  8   =  4
12  %  2   =  0           12  %  9   =  3
12  %  3   =  0           12  %  10  =  2
12  %  4   =  0           12  %  11  =  1
12  %  5   =  2           12  %  12  =  0
12  %  6   =  0           12  %  13  =  12
12  %  7   =  5           12  %  0   =  undefined
```

Do you notice that when dividing into 12, several numbers have a remainder of 0? What does that mean? If I divide two numbers and the *remainder* is 0, it means that one number is a *factor* of the other. This actually is a very useful feature that will be used in future programs. Maybe you are still a little confused. It might be good to take a trip down memory lane -- back to when you first learned about long division. Look at the 5 examples in figure 3.8 below.

Figure 3.8



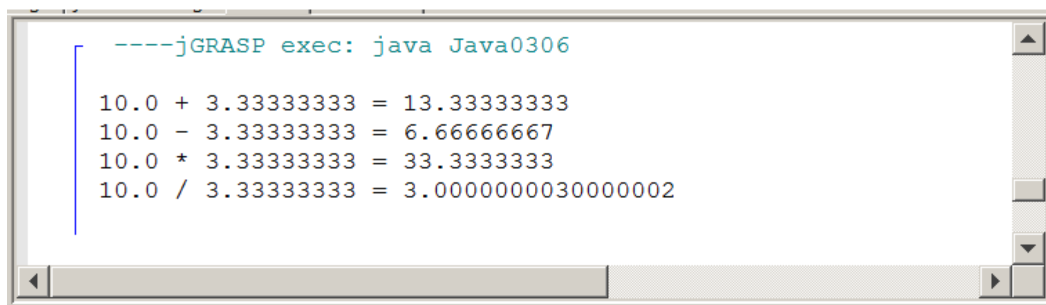
3.4 The double Number Data Types

You saw that Java has four different integer data types. Integers can be declared from as small as 1-byte storage to as large as 8-byte memory allocation. Integers are nice and used for many purposes, but there are also many other computations that require fractions. In science, industry and business, fractions are a way of life. For instance, interest on bank loans and savings accounts are computed as percentages of the principal amount, and percentages must involve computation with fractions.

You probably remember from science that real numbers can be expressed in scientific notation. A real number like 12345.54321 can also be expressed as 1.234554321+04. Note that in the case of the scientific notation, the decimal point moves to a different location or perhaps you can say that the decimal point *floats* to another location. A strange term to you, perhaps, but scientific notation is also known as floating point notation in computer science and real numbers are called **floating point** numbers. Why this long explanation? Without it you may not understand why Java uses the keyword **float** for a real number data type. The next program example, in figure 3.9, shows a second real number data type, called **double**. The reserved word **double** may seem even weirder to you than **float**. The naming actually is quite logical because a **double** variable has twice or double the bytes for memory than a **float** variable. The programs in Exposure Java and the AP Computer Science Examination only use the **double** type.

Figure 3.9

```
// Java0306.java
// This program introduces the real number type <double>.
// This program demonstrates the four real number operations.

public class Java0306
{
	public static void main (String[] args)
	{
		double d1 = 0;
		double d2 = 10.0;
		double d3 = 3.33333333;

		d1 = d2 + d3;
		System.out.println(d2 + " + " + d3 + " = " + d1);
		d1 = d2 - d3;
		System.out.println(d2 + " - " + d3 + " = " + d1);
		d1 = d2 * d3;
		System.out.println(d2 + " * " + d3 + " = " + d1);
		d1 = d2 / d3;
		System.out.println(d2 + " / " + d3 + " = " + d1);
		System.out.println();
	}
}
```



Real number data types, like Java's **float** and **double**, have four arithmetic operators. There are addition, subtraction and multiplication operations, as there were with the integer data types. Division uses the same / operator character, but it performs real number division, provided Java detects the presence of real numbers in the division operation. Later in this chapter we will return to this issue since there are potential problems that need to be addressed to insure that real number division is used when it is desired.

Java textbooks sometimes state that the **%** remainder or mod division does not exist for real numbers. Real numbers do not have remainder division in any practical sense. There also is the issue that Java is based on C++, which does not allow remainder division with real number data types. Many people, who came from C++, myself included, started out assuming that Java followed C++ and that there would be no real number remainder capabilities.

Well that is wrong and I might be tempted to say that it does not exist, but there are always very bright students who are quick to check on the accuracy of a teacher's statement. Java will allow the use of the **%** remainder operator with real numbers. The program will compile and it will actually compute remainder values, but the results usually have no practical value. Personally, I have never used real number remainder division in any program.

Real Arithmetic Operators

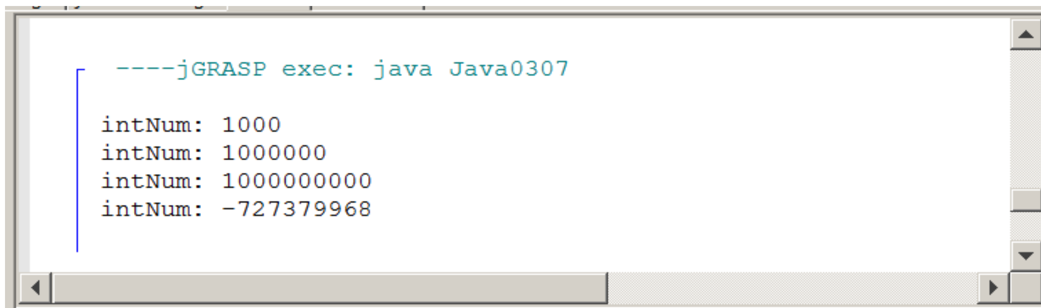
Addition:	6.75 + 2.5	=	9.25
Subtraction:	6.75 - 2.5	=	4.25
Multiplication:	6.75 * 2.5	=	16.875
Real # Quotient Division:	6.75 / 2.5	=	2.7
Rea # Remainder Division:	6.75 % 2.5	=	1.75

3.5 Numerical Representation Limits

You learned earlier that data type selection is important because it saves computer memory. You just saw that Java has four different integer types that range from one-byte to eight-bytes in memory allocation. Perhaps you understand that it is wise to use the smallest possible data type to conserve memory space. Saving memory space is an important goal, but it cannot be at the expense of program accuracy. It is possible to be so stingy with memory usage that mathematical operations do not have enough space to operate correctly. Such a problem is called **memory overflow**, which is demonstrated by the next program example, **Java0307.java**, in figure 3.10. It is the first of several program examples that will demonstrate that mathematical accuracy and computer accuracy are not always equal. Computer numbers are limited by the finite storage of numerical variables. Mathematics has no numerical boundaries. Computers have very specific boundaries for numerical representations.

Figure 3.10

```
// Java0307.java
// This program demonstrates memory overflow problems.
// Saving memory is important, but too little memory can
// also cause problems.

public class Java0307
{
	public static void main (String[] args)
	{
		int intNum = 1000;
		System.out.println("intNum: " + intNum);
		intNum = intNum * 1000;
		System.out.println("intNum: " + intNum);
		intNum = intNum * 1000;
		System.out.println("intNum: " + intNum);
		intNum = intNum * 1000;
		System.out.println("intNum: " + intNum);
	}
}
```



Memory Overflow Problems

Memory overflow is a situation where the assigned value of a variable exceeds the allocated storage space. The resulting value that is stored will be inaccurate and can change from positive to negative or negative to positive.

Avoid memory overflow problems by using a data type that can handle the size of the assigned values. It is important to save computer memory. However, do not be so stingy with memory that overflow problems occur.

Program **Java0307.java** initializes **intNum** to **1000**. The first output displays the initial value of **intNum** and then there are successive displays each time **intNum** is multiplied by **1000**. This works fine for two multiplications. After the third multiplication the output is quite bizarre. This program has a **memory overflow** problem, which can cause very annoying program errors.

To understand what is happening, we need to take a little detour to driving a car. The odometer of a car has a maximum number of miles that can be displayed based on the number of displayed digits. Most odometers have 6 or 7 digits.

Consider a 6-digit odometer with **99,999** miles.

0	9	9	9	9	9

This odometer cannot display a larger number with 5 digits. One more mile and the entire display will change. Each one of the 9 digits will change from **9** to **0**, and the **0** on the far left will become **1.**

1	0	0	0	0	0

You also need to realize that the odometer will max out at **999,999**. It is not possible to display **1,000,000**. The limit of displayed mileage is based on the limit of the number of displayed digits.

The same exact logic applies to variables in a computer. Different variables are assigned different amounts of memory. In Java a **short** integer is allocated **two bytes** of memory. This means a total of **16 bits** are used to represent the **short**

integer in memory. Furthermore, remember that every bit can only have two different values: **0** or **1**. As a human being you may think in base-10, but the computer is busily storing values, and computing values, in base-2 machine code. The largest possible 6-digit number in base-10 is **999,999** and the absolute largest possible 16-digit number in base-2 is represented by the simulated memory below.

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Base 2 number **1111 1111 1111 1111** = **65535** in base-10.

This does not explain why multiplying a positive number times a positive number gives negative results. The largest possible integer, depicted above is not how a **short** (integer) is represented in Java and most other programming languages. Numbers can be positive or negative and the first bit is the ***sign*** bit. That leaves 15 bits for representing the number. The **0** in the first bit indicates a positive number. A **1** in the first bit is for a negative number. This means that **0** followed by **15 1**s is the largest **short** integer, which equals **32,767**.

How Positive Numbers Give Negative Results

0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Now logically speaking **32767** + **1** = **32768**. If we convert these same values to base 2, we get:

0111 1111 1111 1111 + **1** = **1000 0000 0000 0000**

This follows the same logic as **99,999** + **1** = **100,000**. We do not see that logic with the **32767**, base-10 number, because we have not reached any maximum digit values with base-10. At the base-10 level, **32,767** simply increments to **32,768**. At the base-2 level the equivalent values "rollover" and result in the integer value shown below.

1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Note that the sign-bit, which was positive, has changed to **1** and now the whole number is negative. This is as far as we want to look at these negative numbers. There is more complexity to the storing and management of negative numbers.

If you read other computer science textbooks, you may see something called *complement arithmetic*. There is a special way to compute negative values in base-2. Right now the biggest point to realize, *and remember*, is that you will get incorrect values when the variable "overflows." This means that you need to be very careful that you pick the correct data type.

Memory Overflow Problems

Memory overflow is a situation where the assigned value of a variable exceeds the allocated storage space. The resulting value that is stored will be inaccurate and can change from positive to negative or negative to positive.

Avoid memory overflow problems by using a data type that can handle the size of the assigned values. It is important to save computer memory. However, do not be so stingy with memory that overflow problems occur.

Floating point numbers have a more complicated storage mechanism than integers. The first bit is still the sign bit, but then storage is very different. In this introductory course it is important to be aware that there are limitations in the accuracy of computer numbers. Realization of these limitations helps programmers to select the correct data type.

For instance, consider program **Java0308.java**, in figure 3.11, which displays three real numbers. The **double** variables store many digits beyond the decimal point, but this storage is limited, as you can see with the **num3** variable.

Also note that the **num3** besides avoiding several digits that are not displayed also performs a rounding function. The last few digits assigned are **23456789**, but the actual digits stored are **2346**.

Figure 3.11

```
// Java0308.java
// This program shows that there is a memory storage limitation to
// how many digits are stored beyond the decimal point.
// Observe that the last digit displayed is rounded off in num3.


public class Java0308
{
	public static void main (String[] args)
	{
		double num1 = 1.012345;
		double num2 = 1.0123456789;
		double num3 = 1.01234567890123456789;
		System.out.println("num1: " + num1);
		System.out.println("num2: " + num2);
		System.out.println("num2: " + num3);
		System.out.println("\n\n");
	}
}
```

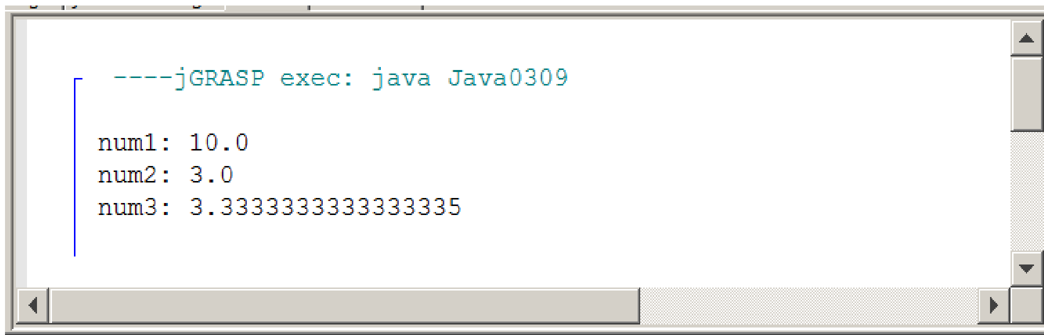
```
----jGRASP exec: java Java0308

num1: 1.012345
num2: 1.0123456789
num2: 1.0123456789012346
```

When Java performs computations, there is a limit to how far the computations are performed. You have already noticed that there is a definite limit to the number of digits that can be displayed beyond the decimal point.

Program **Java0309.java**, in figure 3.12, shows a consequence of computer calculations. Number **10.0** is divided by number **3.0** and the expected answer should be **3.33333333333333333333** for as far as the computer can display the numbers. However, there is a **5** at the end, which is quite surprising. This is serious evidence that computers can create round-off errors in computation.

Figure 3.12

```
// Java0309.java
// This program demonstrates another error.
// The program output displays a number that is mathematically incorrect.

public class Java0309
{
	public static void main (String[] args)
	{
		double num1 = 10.0;
		double num2 = 3.0;
		double num3 = num1 / num2;

		System.out.println("num1: " + num1);
		System.out.println("num2: " + num2);
		System.out.println("num3: " + num3);

		System.out.println("\n\n");
	}
}
```



3.6 Arithmetic Shortcut Notations

The language **C** started a shortcut trend with operators. This trend continued with C++, and Java adopted the popular shortcuts founded by the older C programming language. Shortcuts are popular with programmers and less popular with teachers. It is possible to create confusing code with C shortcuts and this section will show you the available shortcuts and give you advice in both good and bad programming habits. Program **Java0310.java** demonstrates the Java **unary** operators, which are operators with a single operand.

This can look strange to you, because in your previous exposure to mathematical notation you probably only saw binary operators. The program example, in figure 3.13, shows that unary operators can be used with **postfix** or **prefix** style.

It may seem odd to provide two styles to accomplish the same goal, but the reality is that there is a subtle, but very significant difference between the postfix and prefix style. Check out the next program and see if the difference makes sense. You need to look carefully at every output line in the program and compare the output with the corresponding program statement. Failure to understand the difference between prefix and postfix notation can create logic errors.

Figure 3.13

```
// Java0310.java
// This program shows "unary" arithmetic shortcut notation in Java.
// Note that "postfix" x++ and "prefix" ++x do not always have the same result.


public class Java0310
{
	public static void main (String[] args)
	{
		int num = 10;
		System.out.println("num equals " + num);
		num++;
		System.out.println("num equals " + num);
		++num;

		System.out.println("num equals " + num);
		System.out.println("num equals " + num++);
		System.out.println("num equals " + num);
		System.out.println("num equals " + ++num);
		System.out.println("num equals " + num);
		System.out.println();
	}
}
```

```
----jGRASP exec: java Java0310

num equals 10
num equals 11
num equals 12
num equals 12
num equals 13
num equals 14
num equals 14
```

Both ++**num;** and **num**++**;** have the same meaning, which is **num** = **num** + **1;** The same is true with **--num;** and **num--,** which is the same as **num** = **num - 1;** Now look at the output of program **Java0310.java**. The value of **num** starts out with **10** and then is incremented by **1** twice with the unary ++ operator. Now it gets a little tricky and the operation is placed inside an output statement. Incrementing **num** by **1** and displaying the value of **num** are done in the same statement. This provides a dilemma for the computer. Should the computer first display the value of **num** and then increment the value or should the computer do the process in reverse? The answer is both. Yes both are true. In the case of the postfix operator, **num**++, the value of **num** is first displayed and then incremented. **--num**, on the other hand, first decrements **num** and then displays the value. This type of code can easily create logic errors.

Java Unary Operators

k++;	is the same as:	**k = k + 1;**
++k;	is the same as:	**k = k + 1;**
k--;	is the same as:	**k = k - 1;**
--k;	is the same as:	**k = k - 1;**

Proper Usage:

```
k++;
System.out.println(k);
--k;
System.out.println(k);
```

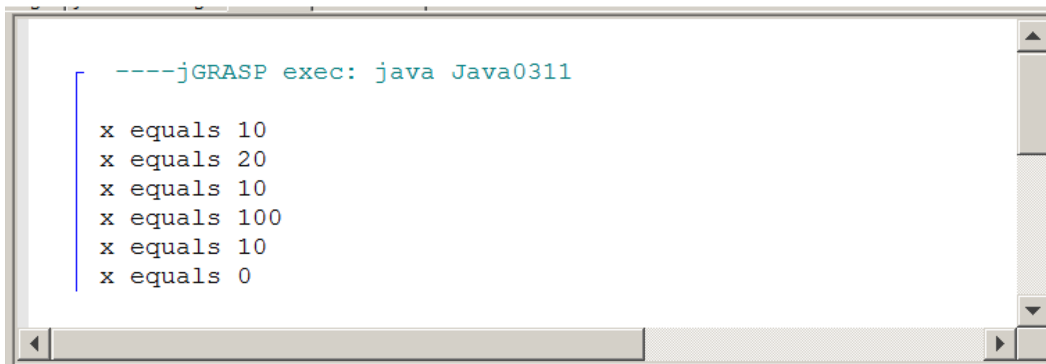
Problematic Usage:

```
System.out.println(k++);
System.out.println(--k);
```

Unary operators are lovely, but they are quite limited. Incrementing by one or decrementing by one can be quite boring. Sometimes you want to go for broke and increment by two, three or maybe even ten. Are there shortcuts for such type of operations? Binary operators have shortcuts, and like unary shortcuts, there are potential pitfalls where program statements can be quite ambiguous. Shortcuts are good, but there is such a thing as too much of a shortcut and this can make a program difficult to debug, comprehend and update. This is a warning and a later program will demonstrate how confusing it can be to use too many shortcuts. Right now examine **Java0311.java** in figure 3.14 and observe the shortcut syntax of binary operations.

Figure 3.14

```
// Java0311.java
// This program shows arithmetic assignment operations in Java.
// x+=10; is the same as x = x + 10;

public class Java0311
{
	public static void main (String[] args)
	{
		int x = 10;
		System.out.println("x equals " + x);
		x += 10;
		System.out.println("x equals " + x);
		x -= 10;
		System.out.println("x equals " + x);
		x *= 10;
		System.out.println("x equals " + x);
		x /= 10;
		System.out.println("x equals " + x);
		x %= 10;
		System.out.println("x equals " + x);
		System.out.println();
	}
}
```

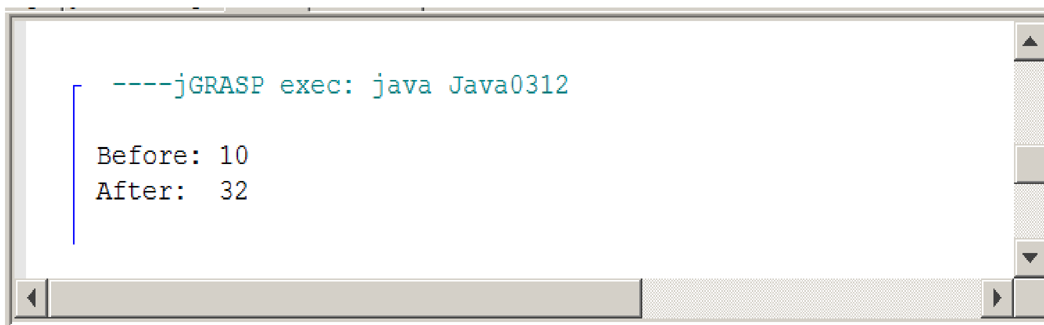


Program **Java0311.java** did not seem so bad. None of the dire shortcut warnings appeared to be visible. Everything executed just nicely as expected, and it was done with less program code, courtesy of the clever shortcuts. So just what is this ambiguous stuff I am talking about?

The next program example is a serious *no-no*. Teachers will lobby for Singapore-style caning privileges if you dare program in this style. There are two important points to be made when you look at program **Java0312.java**, in figure 3.15. First, it is rather amazing that the compiler can digest this glob, and second how on Earth do you know what the output will be?

Figure 3.15

```
// Java0312.java
// This program demonstrates very bad programming style by
// combining various shortcuts in one statement.  It is difficult
// to determine what actually is happening.


public class Java0312
{
	public static void main (String[] args)
	{
		int x = 10;
		System.out.println("Before: " + x);
		x += ++x + x++;
		System.out.println("After:  " + x);
		System.out.println();
	}
}
```



Do you have a clue why the value of **x** equals **32** at the conclusion of this monstrosity? You do not have any idea? There actually is a logical sequence of steps that will explain the output. The point is that it looks confusing, it is confusing and it is an awful way to create a program.

Binary Operator Shortcuts	
No Shortcut Notation	**Shortcut Notation**
k = k + 5 **k = k - 5** **k = k * 5** **k = k / 5** **k = k % 5**	**k += 5** **k -= 5** **k *= 5** **k /= 5** **k %= 5**

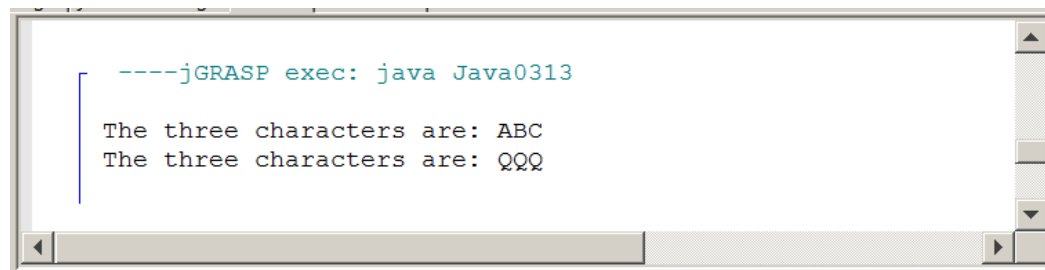
3.7 The char and String Data Types

The very first Java program you saw in this book used only strings. Strings are extremely common. It is a string of characters that forms a word, and it is a string of words that forms a sentence. Java processes characters and strings with two data types. There is the **char** data type for processing individual characters, and there is the **String** data type for processing sets of one or more characters.

In previous programs you have observed that a string of characters is contained between double quotes. That is still very true. There is a small difference for a single character, which needs to be contained between two single quotes. Program **Java0313.java**, in figure 3.16, starts by concentrating on the humble **char** data type. Three different character variables are declared and initialized. This program also demonstrates that *chain assignment* or *chaining* is possible. This is another type of shortcut. In a single program statement the character **'Q'** is assigned to all three variables.

Figure 3.16

```
// Java0313.java
// This program demonstrates the <char> data type.
// It also demonstrates how assignment can be "chained" with
// multiple variables in one statement.

public class Java0313
{
	public static void main (String[] args)
	{
		char c1 = 'A';
		char c2 = 'B';
		char c3 = 'C';
		System.out.println("The three characters are: " + c1 + c2 + c3);
		c1 = c2 = c3 = 'Q';
		System.out.println("The three characters are: " + c1 + c2 + c3);
		System.out.println();
	}
}
```

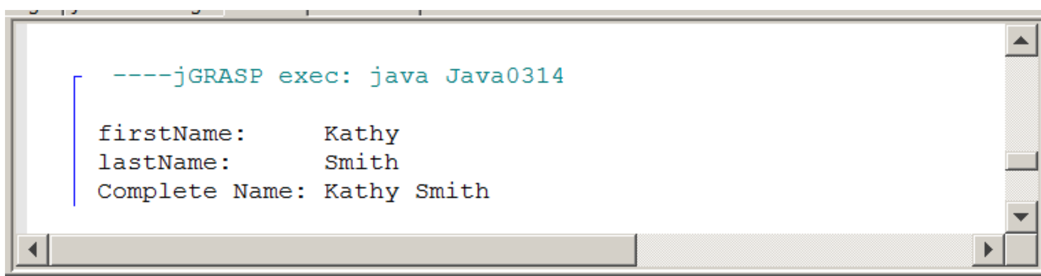


If excitement is your goal in life then **char** is not going to provide much. It is pretty much a dull data type. String is a great deal more interesting. Now you might argue that excitement has been seriously lacking from all this variable stuff. Like, where are the spaceships shooting photon torpedoes? Well those photon torpedoes are used a lot in a variety of Star Trek episodes and students in an Advanced Graphics Programming course might wish to create a program that performs that type of sophistication.

Right now your excitement revolves around variables. The topic at hand is the **String** data type and watch with amazement as you see the next program **Java0314.java**, in figure 3.17, combine various string variables together in seamless perfection.

Figure 3.17

```
// Java0314.java
// This program demonstrates the <String> data type.

public class Java0314
{
	public static void main (String[] args)
	{
		String firstName = "Kathy" ;
		String lastName = "Smith";
		System.out.println("firstName:          " + firstName);
		System.out.println("lastName:           " + lastName);
		System.out.println("Complete Name:   " + firstName + "  " + lastName);
		System.out.println();
	}
}
```



Did you observe that strings are performing "addition" here. At least the plus operator is used and it seems that some type of adding is going on. There is a form of addition shown here that is peculiar to strings, and a lovely name exists for this operation, which is known as *concatenation*. This is an example of *overloading* the plus operator. The same exact operator performs totally different functions with numbers and with strings.

String Concatenation

Concatenation is the appending of a 2nd string to a 1st string.

```
"Hello" + "World"  =  "HelloWorld"
"Hello" + " " + "World"  =  "Hello World"
"100" + "200"  =  "100200"
```

The plus operator (**+**) is used both for arithmetic addition and string concatenation. The same operators perform two totally different operations, called **overloading**.

3.8 The boolean Data Type

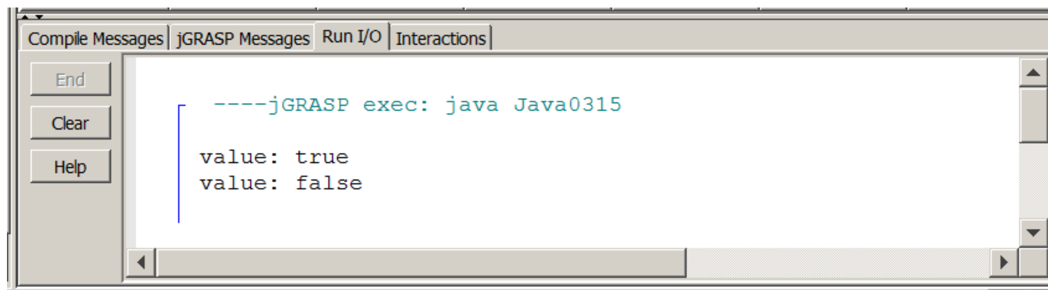
More than a century ago there was a mathematician, George Boole, who developed a new branch of mathematics. His mathematics did not involve arithmetic nor Algebra, but logical statements that are either **true** or **false**. This new branch of mathematics was named *Boolean Algebra* after its founder. Today, in computer science, a data type that has only two values of **true** and **false** is called a *Boolean* data type, and in Java you use the reserved word, **boolean**.

Two values may not seem to have any practical purpose. What can you do with true and false? A very nice feature is readability. You will find that programs repeat code, like repeating until a password is correctly entered. In such a case a variable can be called **correctPassword** and you can repeat until **correctPassword**. Another example is a search routine, which keeps repeating until **found**.

Program example **Java0315.java**, in figure 3.18 demonstrates the proper syntax to declare a Boolean variable, but it does not explain how to use *Boolean* variables. The **boolean** data type is included here to complete the simple data types. You will learn in later chapters how to actually use this very unique data type in practical programs.

Figure 3.18

```
// Java0315.java
// This program demonstrates the <boolean> data type.
// The boolean type can only have two values: true or false.

public class Java0315
{
	public static void main (String[] args)
	{
		boolean value = true;
		System.out.println("value: " + value);
		value = false;
		System.out.println("value: " + value);
		System.out.println();
	}
}
```



AP Computer Science Examination Alert

Only the **int**, **double**, **boolean** and **String** data types will be tested on the AP Computer Science Examination.

3.9 Declaring Constants

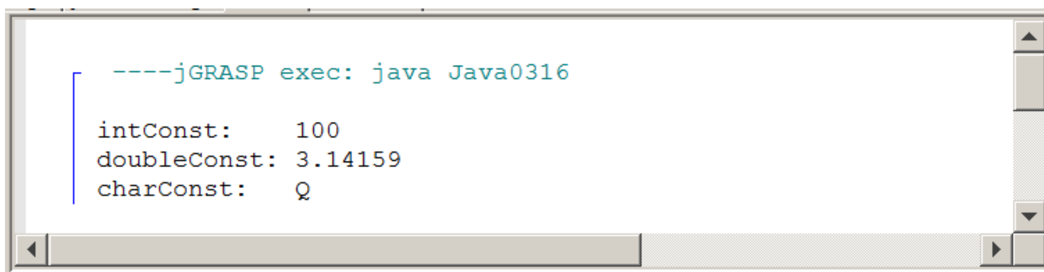
We are done with simple data types. In more formal language Java's simple data types are called *primitive data types*. You have seen them all and they will provide a base for many of your programs in the beginning of this course. But you are not done with this chapter. There are a few related topics that link to

simple data types. So far all the data types were used with a variable declaration and variable implies that some initialized value is able to change or vary.

Now what if you want to store a value somewhere in memory for a specified data type, but you do not want the value to change? If you write a program that computes a variety of areas and volumes that involve curves, you will need to use **PI**. Now do you want the value of **PI** to change? Hardly, **PI** is a classic example of a constant. Java allows you to create programs with identifiers that store values, almost the same as variables, but with some minor change the variable is now a constant, as demonstrated by program **Java0316.java**, in figure 3.19.

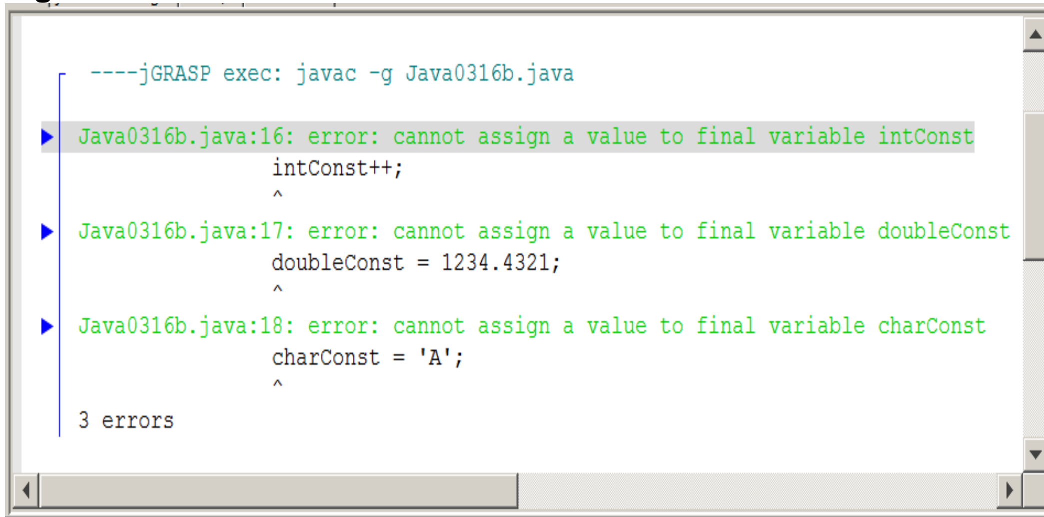
Figure 3.19

```
// Java0316.java
// This program demonstrates how to create "constant" identifier
// values with the <final> keyword.
// Removing the comments from the three assignment statements
// will result in compile errors.

public class Java0316
{
	public static void main (String[] args)
	{
		final int intConst = 100;
		final double doubleConst = 3.14159;
		final char charConst = 'Q';
//		intConst++;
//		doubleConst = 1234.4321;
//		charConst = 'A';
		System.out.println("intConst:       " + intConst);
		System.out.println("doubleConst: " + doubleConst);
		System.out.println("charConst:     " + charConst);
		System.out.println();
	}
}
```



You may feel that **Java0316.java** is no different from many of the programs shown in this chapter. There is some odd-looking **final** keyword thrown in, but the output is no different than anything you saw with variables. You do have a good observation and the program contains a feature to satisfy your curiosity. Notice how three lines are commented out. Each one of these three lines is meant to change the initial values of the **intConst**, **doubleConst** and **charConst** identifiers. Remove the comments and recompile the program. You will not get very far. The Java compiler is most displeased that you do not understand the seriousness of the situation. Look at the error message shown in figure 3.20. Personally, I think the error message is an oxymoron. It states *cannot assign a value to final variable A*. If you cannot change the value then it is not a variable. Java has decided to call this feature a **final variable**. I am more comfortable with the term **constant** that is used in other program languages.

Figure 3.20



```
----jGRASP exec: javac -g Java0316b.java

Java0316b.java:16: error: cannot assign a value to final variable intConst
                intConst++;
                ^
Java0316b.java:17: error: cannot assign a value to final variable doubleConst
                doubleConst = 1234.4321;
                ^
Java0316b.java:18: error: cannot assign a value to final variable charConst
                charConst = 'A';
                ^
3 errors
```

3.10 Documenting Your Programs

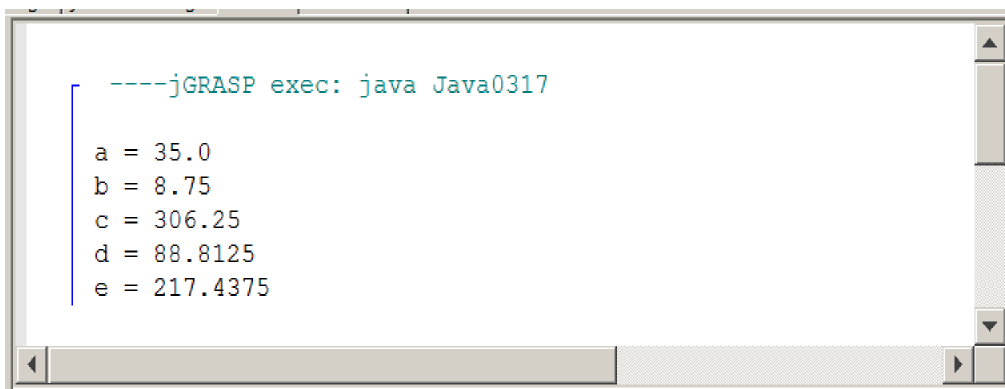
Program documentation is a major big deal. Perhaps to you it is a big deal because some irritating computer science teacher keeps after you to document your programs.

You will not fully appreciate the need for documentation in a first year course. Once the programs you write reach a certain size, it is not possible to test, debug or alter such programs without proper documentation.

The first form of program documentation is to use comments. You were shown how to create *single-line comments* and *multi-line comments* back in Chapter 2. When a program uses variables, another form of program documentation is possible. To illustrate the need for program documentation the next program, **Java0317.java**, shown in figure 3.21, has no program documentation whatsoever. When you look at this program, do you have any clue what it does? Even the program's output, tells you nothing about what is happening with this program.

Figure 3.21

```
// Java0317.java
// This is an example of a poorly written program with single-letter variables.
// Do you have any idea what this program does?

public class Java0317
{
	public static void main (String[] args)
	{
		double a;
		double b;
		double c;
		double d;
		double e;
		a = 35;
		b = 8.75;
		c = a * b;
		d = c * 0.29;
		e = c - d;
		System.out.println("a = " + a);
		System.out.println("b = " + b);
		System.out.println("c = " + c);
		System.out.println("d = " + d);
		System.out.println("e = " + e);
		System.out.println();
	}
}
```

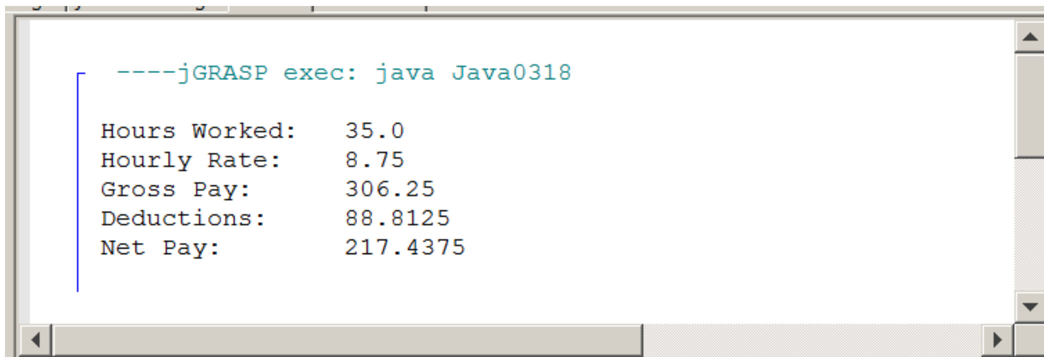


Program **Java0317.java** makes no sense because it uses *single-letter variables*. Several decades ago, this actually was the proper way to program. Back in the 1960s and 1970s computer memory was scarce and very expensive. Programmers had to do anything they could save every byte possible. This was so extreme that when a year needed to be stored, they would only store the last 2 digits. For example, **1968** was simply stored as **68**. This is what led to the whole *Y2K* mess just before we hit the *Year 2000*. (Ask your teacher about Y2K)

Today computer memory is abundant and very cheap. The need to save every byte possible no longer exists. Program *readability* is now a big issue. This is another part of program documentation. Program **Java0318.java**, shown in figure 3.22, does the exact same thing as the previous program. There is only one difference. The variables have different names now. Does the program make more sense now?

Figure 3.22

```
// Java0318.java
// This program does exactly the same thing as the previous program.
// By using self-commenting variables, the program is much easier to read and understand.

public class Java0318
{
	public static void main (String[] args)
	{
		double hoursWorked;
		double hourlyRate;
		double grossPay;
		double deductions;
		double netPay;

		hoursWorked = 35;
		hourlyRate = 8.75;
		grossPay = hoursWorked * hourlyRate;
		deductions = grossPay * 0.29;
		netPay = grossPay - deductions;
		System.out.println("Hours Worked:      " + hoursWorked);
		System.out.println("Hourly Rate:       " + hourlyRate);
		System.out.println("Gross Pay:         " + grossPay);
		System.out.println("Deductions:        " + deductions);
		System.out.println("Net Pay:           " + netPay);
		System.out.println();
	}
}
```



Program **Java0318.java** should have made more sense because the variables are now *self-commenting*. A *self-commenting variable* is a variable whose name describes what the variable is used for. In the previous program the variable for *Net Pay* was **e**. In this program, the variable for *Net Pay* is **netPay**. This is why this program is so much easier to read and understand. The age of single-letter variables is gone. Variables should now be *words* like **deductions** or *compound words* like **hoursWorked**.

Earlier, it was mentioned that *comments* are part of program documentation. Just because your program has variables which are *self-commenting* does not mean there is no need for well-placed comments in your program. At the start of a program you need to use a heading that explains some general information about the program. At this place it makes sense to use a *multi-line comment*. There are other places in the program were a quick *single-line comment* provides some needed information.

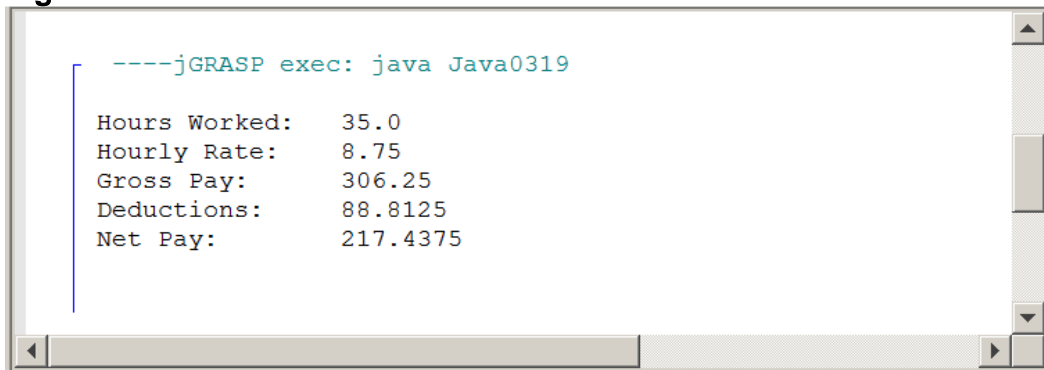
Program **Java0319.java**, in figure 3.23, demonstrates both types of comments. In particular, note how the comments extend the meaning of the self-commenting variables. For instance, the identifier **hoursWorked** is descriptive, but it is the comment which explains that it means the number of hours worked per week.

Figure 3.23

```
// Java0319.java
// This program adds a multi-line comment at the beginning to help explain the program.
// Several single-line comments are also added to provide more detail for each variable.


/*******************************************************************
 **                                                               **
 **   Payroll Program                                             **
 **   Written by Leon Schram 04-07-14                             **
 **                                                               **
 **   This program takes the hours worked and hourly rate of      **
 **   an employee and computes the gross pay earned.              **
 **   Federal deductions are computed as 29% of gross pay.        **
 **   Finally the take-home pay or net pay is computed by         **
 **   subtraction deductions from gross pay.                      **
 **                                                               **
 ******************************************************************/



public class Java0319
{
	public static void main (String[] args)
	{
		double hoursWorked;		// hours worked per week
		double hourlyRate;		// payrate earned per hour
		double grossPay;		// total earnings in a week
		double deductions;		// total federal tax deductions
		double netPay;			// employee take-home pay
```

```
        hoursWorked = 35;
        hourlyRate = 8.75;
        grossPay = hoursWorked * hourlyRate;
        deductions = grossPay * 0.29;
        netPay = grossPay - deductions;

        System.out.println("Hours Worked:       " + hoursWorked);
        System.out.println("Hourly Rate:        " + hourlyRate);
        System.out.println("Gross Pay:          " + grossPay);
        System.out.println("Deductions:         " + deductions);
        System.out.println("Net Pay:            " + netPay);
        System.out.println();

    }
}
```

Figure 3.23 Continued



3.11 Mathematical Precedence

Java may not use all the exact same symbols for mathematical operations, but the precedence of operations is totally identical. Rules like *multiplication/division before addition/subtraction* and *parentheses before anything else* apply in Java. Parentheses are also used in the same manner as they are in mathematics. You do need to be careful that operators are always used. In mathematics, operators are frequently assumed, but not used. This is especially true for the multiplication operator. A small chart in figure 3.24 helps to clarify this point.

Figure 3.24

Be Aware of Hidden Operators in Mathematics	
Mathematics	**Java Source Code**
5XY	**5*X*Y**
4X + 3Y	**4*X + 3*Y**
6(A – B)	**6*(A – B)**
$\frac{5}{7}$	**5.0 / 7.0**
$\frac{A + B}{A - B}$	**(A + B) / (A – B)**
$\frac{AB}{XY}$	**(A * B) / (X * Y)**

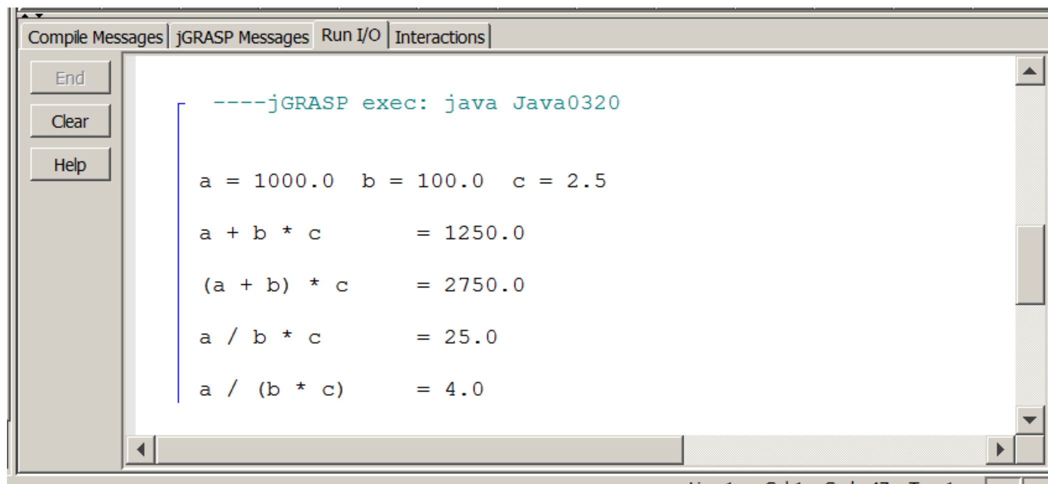
Mathematical precedence usually is not a problem for students. However, leaving out operators or parentheses, which are not required in regular mathematical expressions, is a common problem for beginning computer science students.

Program **Java0320.java**, in figure 3.25 demonstrates a variety of expressions that use mathematical precedence. You will note that 2 expressions can be very similar, but yield very different results just because one has a strategically placed set of parentheses ( ).

Figure 3.25

```
// Java0320.java
// This program demonstrates mathematical precedence in Java operations.

public class Java0320
{
	public static void main (String[] args)
	{
		double a, b, c, result;
		a = 1000;
		b = 100;
		c = 2.5;

		System.out.println();
		System.out.println("a = " + a + "  b = " + b + "  c = " + c);
		result = a + b * c;
		System.out.println("\na + b * c   = " + result);
		result = (a + b) * c;
```

```
        System.out.println("\n(a + b) * c = " + result);
        result = a / b * c;
        System.out.println("\na / b * c   = " + result);
        result = a / (b * c);
        System.out.println("\na / (b * c) = " + result);
        System.out.println();
    }
}
```

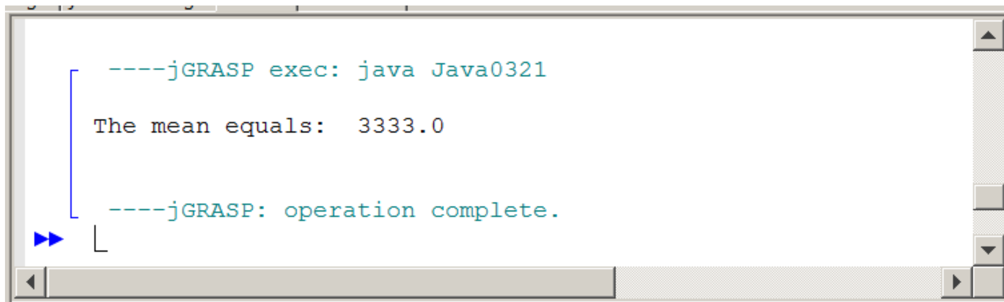


3.12 Type Casting

Java does not get confused if you always make sure that you assign the correct data type to a variable. All the program examples in this chapter, up to this point, have carefully assigned correct data values. Now what happens if you are not very careful? Will the Java compiler get excited or simply ignore the problem. Program **Java0321.java**, in figure 3.26, is an average program. Three numbers, which are declared and initialized as integers, need to be averaged. The resulting average needs to be assigned to **mean**, which is **double**. Will this program work correctly or will Java get confused?

Figure 3.26

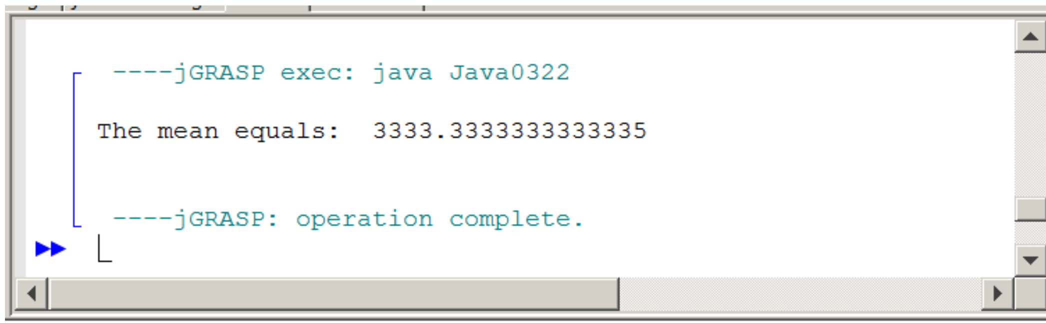
```
// Java0321.java
// This program demonstrates that the intended computation may not be
// performed by Java.  The expression on the right side of the assignment
// operator is performed without knowledge of the type on the left side.


public class Java0321
{
	public static void main (String[] args)
	{
		int nr1 = 1000;
		int nr2 = 3000;
		int nr3 = 6000;
		double mean;
		mean = (nr1 + nr2 + nr3) / 3;
		System.out.println("The mean equals:  " + mean);
		System.out.println();
	}
}
```



Java is not at all confused. Computers are never confused, because they follow instructions faithfully. The assignment statement on the right side of the equal sign shows three integer variables that need to be added and then divided by the integer 3. In other words, everything on the right side is integer business. It should come as no shocking surprise that Java treats the expression as an integer computation and performs *integer division*. This is not the desired computation. There is no need to argue that **mean** is a **double**. Java computes the expression on the right side based on the information supplied on the right side.
There is a solution to this problem that will communicate to the computer what your intentions are. The solution is called ***type casting***, which is a major topic in the Java programming language. Program **Java0322.java**, in figure 3.27, is almost identical to the previous program. The only difference is the **(double)** keyword placed with the statement that computes the **mean**. The result is that Java now knows that the intended division is real number division.

Figure 3.27

```
// Java0322.java
// This program corrects the logic error of Java0321.java.
// Type casting is used to "force" real number quotient division.

public class Java0322
{
	public static void main (String[] args)
	{
		int nr1 = 1000;
		int nr2 = 3000;
		int nr3 = 6000;
		double mean;
		mean = (double) (nr1 + nr2 + nr3) / 3;
		System.out.println("The mean equals:  " + mean);
		System.out.println();
	}
}
```



With program **Java0323.java**, in figure 3.28, you will see the results of various variables type-casted to another data type. The output display is not exactly shocking.

The **int** variable **65** becomes **65.0** when casted to a **double** type.

The **int** variable **65** becomes letter **A** when casted to a **char** type.

The **double** variable **70.1** becomes **70** when casted to an **int** type.

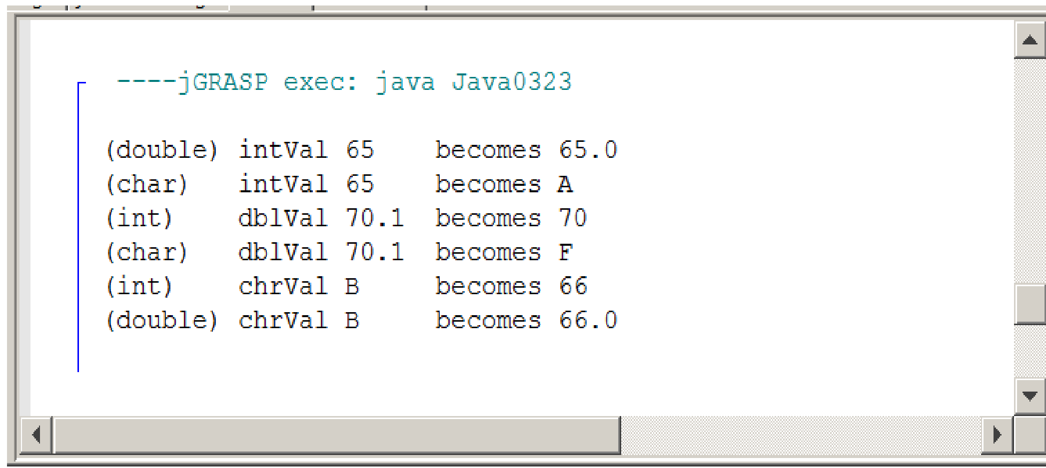
The **double** variable **70.1** becomes **F** when casted to a **char** type.

The **char** variable **B** becomes **66** when casted to an **int** type

The **char** variable **B** becomes **66.0** when casted to a **double** type.

Figure 3.28

```
// Java0323.java
// This program demonstrates "type casting" between different data types.

 public class Java0323
{
	public static void main (String args[])
	{
		int intVal     = 65;
		double dblVal = 70.1;
		char chrVal  = 'B';
		System.out.println("(double)  intVal 65     becomes " + (double) intVal);
		System.out.println("(char)    intVal 65     becomes " + (char) intVal);
		System.out.println("(int)     dblVal 70.1  becomes " + (int) dblVal);
		System.out.println("(char)    dblVal 70.1  becomes " + (char) dblVal);
		System.out.println("(int)     chrVal B      becomes " + (int) chrVal);
		System.out.println("(double)  chrVal B      becomes " + (double) chrVal);
		System.out.println();
	}
}
```

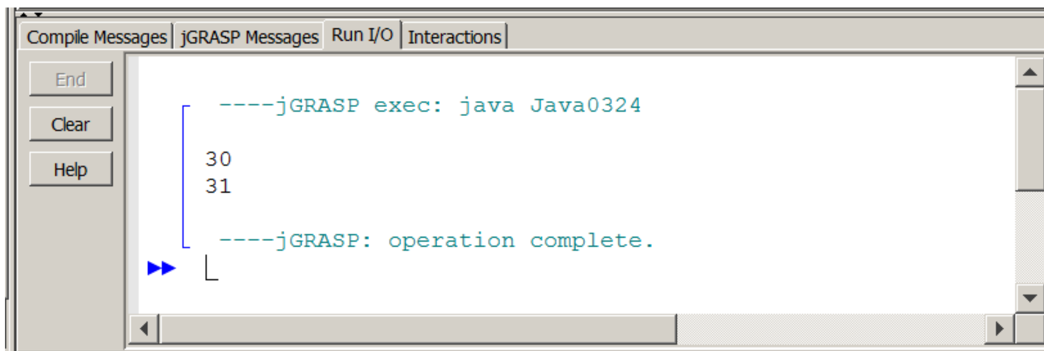


In programming order of operations is not restricted to mathematical precedence. You have already seen that **++q** and **q++** when combined with a **println** call behave differently. The same is true with type casting.

Program **Java0324.java**, in figure 3.29, shows two program statements that use perform a cast to **(int)**. The outputs are different. In the first statement the value of **3.14159** is first cast to **3** and then multiplied by **10**, In the second statement the parentheses take priority and the multiplication executes first, followed by the casting operation.

Figure 3.29

```
// Java0324.java
// Sequence of operations matters.
// Casting executes before multiplication
// unless parentheses are used.


public class Java0324
{
   public static void main (String[] args)
   {
      int a = (int) 3.14159 * 10;
      int b = (int) (3.14159 * 10);
      System.out.println(a);
      System.out.println(b);
   }
}
```



In future chapters you will see this precedence happening in other situations as well. Failure to realize this certainly causes program logic errors. Keep in mind that Java is clueless. The syntax is correct, with or without parentheses. Java is pleased to compile your program either way.

You can only catch logic errors by carefully testing your program with different variable values. It is not simply a matter of having correct programs. There are mean, tricky teachers who enjoy giving you questions on this very issue and some of these teachers create AP Computer Science questions. Be careful, the error is subtle and easily overlooked.

3.13 Summary

This chapter introduced the Java *simple data types*. A simple data type is simple because it stores a single value in memory. Simple data types are also called *primitive data types*. Program examples were shown that declared variables of a specified data type. Declaring a variable allows the compiler to allocate memory for the value to be stored.

Java has an **int** data type, which occupies four bytes of memory. There are other integer data types available which use more memory for larger integer ranges, or less memory for smaller integer ranges. Java provides five operators for integers: *addition*, *subtraction*, *multiplication*, *integer-division* and *modulus-division* or *remainder-division*.

There are two different real number data types: **float** and **double**. Float is short for *floating point* number and uses four bytes of memory. Double uses eight bytes of memory. The word **double** indicates that this data type has twice the number of bytes than a **float** data type. Java provides four operators for real numbers: *addition*, *subtraction*, *multiplication* and *real number quotient division*.

The specified data type is good for memory efficiency, but stingy use of memory can result in memory overflow. If a value is larger than the space reserved in memory, the result is incorrect, even if the mathematics is flawless.

In Java there are many shortcut notations for both unary operators and binary operators. Every arithmetic operator can be expressed in a shortcut notation. Keep in mind that multiple shortcut operations in the same statement can create very ambiguous program statements that are difficult to predict.

Java can declare character and string variables. The plus operator is used for arithmetic addition with numbers and concatenation with strings. Concatenation means that a string is appended at the end of another string.

This chapter also introduced the **boolean** data type. This data type can store the value **true** or the value **false**. Boolean is included with this chapter to make the chapter complete with all the available simple data types.

Java has a peculiar variable, called a **final** variable that cannot change. I prefer to call this a *constant*. Declaring a constant is identical to declaring a variable with the reserved **final** in front of the data type.

Java programs use the same mathematical precedence that is used in mathematical computation. In mathematics there are implied operations that need to be explicitly shown in a Java program. It is sufficient to state **AB** + **CD** in mathematics. In Java such an expression needs to be **A*B** + **C*D**.

It is possible to alter data types with *type casting*. **Int**egers can become **double**s and **char**acters. **Double**s can become **int**egers and **char**acters. **Char**acters can become **int**egers and **double**s. Type casting is achieved by placing the new, desired data type inside parentheses in front of the variable to be altered.

Java will perform a cast before a mathematical operations. Be careful to use proper parentheses if the mathematical operations must execute first.

Only the **int**, **double**, **boolean** and **String** data types will be tested on the AP Computer Science Examination. The **byte**, **short**, **long**, **float** and **char** data types will NOT be tested.