Chapter 4
Variables and Constants

Variables and constants are two important concepts explained in this chapter. Using primitive and abstract data types to declare variables is covered. User input, numeric expressions, and assignment operators are also discussed.

Declaring Variables

A *variable* is a name for a value stored in memory. Variables are used in programs so that values can be represented with meaningful names. For example, when a variable named `length` is used in a program, it is clear that its value is a distance. Variables should be used to represent values because they make code easier to read, understand, and modify.

declaration

A variable must be declared before it is used. A *declaration* takes the form:

```
<type> <name>
```

data type

identifier

The declaration includes two parts. The first is the *data type*, which determines the type of data the variable will store. The second part of a declaration is the variable name, called the *identifier*. For example, in the declaration

```
int length;
```

`int` is the data type and `length` is the identifier. An `int` stores an integer value, which is a positive or negative whole number. When an integer variable is declared it stores the value 0.

An identifier must begin with a letter and contain only letters, numbers, and some special characters. Typically variable identifiers begin with a lowercase letter. Any word after the first in a variable identifier should begin with an uppercase letter. For example, `rectangleLength`. This code convention allows variables to be easily recognized.

Multiple variables with the same data type can be declared in a single statement, similar to:

```
int length, width;
```

Grouping variables together in a single statement is good programming style when the variables represent related items. Declarations should not be grouped together in the same statement just because the variables are all the same type.

Using Variables

Applications typically contain many variables, as in RectangleArea:

```
/**
 * Calculates and displays the area of a rectangle
 */
public class RectangleArea {

  public static void main(String[] args) {
      int length = 10;    //longer side of rectangle
      int width = 2;      //shorter side of rectangle
      int area;           //calculated area of rectangle

      area = length * width;
      System.out.println("Area of rectangle: " + area);
  }
}
```

RectangleArea produces output similar to:

```
Area of rectangle: 20
```

Variable declarations should be grouped at the beginning of a method. A blank line after the declarations makes it easy to determine where the declarations end.

assignment

equal sign

literal

The value of a variable is changed through assignment. An *assignment statement* is formed with the variable name on the left side of an equal sign and the value it is to receive on the right side of the equal sign. The *equal sign* (=) is an operator that indicates that the variable on the left is to receive the value on the right. The value on the right can be a *literal*, which is any actual value. It could also be another variable or an expression. For example, `area` was assigned the value of the length multiplied by the width (`length * width`). Note that the * symbol indicates multiplication.

initialize

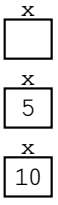
An assignment statement can be part of a variable declaration. In addition to being declared, the variable is *initialized*. For example, in RectangleArea, variables `length` and `width` were assigned values when declared.

concatenation

A `System.out.println()` statement can be used to output the value of a variable. Variable identifiers are not enclosed by quotation marks. To append, or *concatenate*, the value of a variable to a string, the + operator is used. The + operator converts the value of the variable to a string and then concatenates the strings before output.

It is important to realize that a variable can store only one value at any one time. For example, after the following statements execute

```
int x;
x = 5;
x = 10;
```



the value of `x` is 10 because this was the last value assigned to `x`.

Review: RectanglePerimeter

Create a RectanglePerimeter application that calculates and displays the perimeter of a rectangle with width 4 and length 13. The perimeter of a rectangle is calculated as 2w + 2l. Use variables as appropriate.

Primitive Data Types

The `int` data type is called a primitive data type. A variable that is defined with a *primitive data type* stores a single piece of data. Java supports several primitive data types, including:

Type	Bytes	Data Range
`int`	4	a positive or negative integer from –2,147,483,648 to 2,147,483,647
`double`	8	a positive or negative number that may contain a decimal portion in the range –1.7E+308 to 1.7E+308
`char`	2	a single character
`boolean`		`true` or `false`

String Data

Strings are comprised of a set of characters and therefore cannot be represented by a primitive data type. The String class in the java.lang package is used to create string variables. Strings are discussed in more detail in Chapter 6.

An `int` variable uses 4 bytes of memory to store its value and is used for representing whole numbers.

floating point

Values that are represented by the `double` data type are sometimes referred to as *floating point*, meaning that the values contain numbers after the decimal point. Because of the many digits that are possible in a `double`, a variable of this type uses 8 bytes of memory.

TIP Java also supports the `byte`, `short`, `long`, and `float` primitive data types.

A `char` variable requires 2 bytes of memory because Java uses the 16-bit Unicode character encoding. Assignment to a `char` variable requires enclosing a single character in single quotation marks, as in `'a'`.

Variables that are type `boolean` can have only one of two values—`true` or `false`. Boolean variables are particularly useful for representing yes/no or on/off values.

choosing data types

When choosing a data type, it is important to choose the most appropriate type for the quantity being represented. If a value could possibly have a decimal portion, then `double` is the best choice. If a variable will represent only whole numbers, then `int` is the best choice even though `double` will work. Using the most appropriate data types for variables has two benefits. First, both the compiler and the reader will understand the possible values for a variable. Second, the compiler allocates the appropriate memory for the variable.

TIP Primitive data types are also called built-in data types.

Review: Distance – part 1 of 2

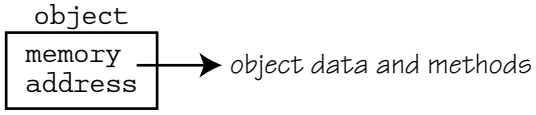
Create a Distance application that calculates and displays the total distance of a race with three segments. The first segment is 12.2 km, the second is 10.6 km, and the third is 5.8 km. Use variables of the appropriate type.

Abstract Data Types

In addition to primitive data types, a variable can be declared using an abstract data type. One kind of *abstract data type* is the class. Many classes are provided in Java, and many more classes will be created throughout this text. Each class defines not just a single piece of data like a primitive data type, but a set of data along with methods for performing actions on that data.

object

A variable declared with a class is called an *object*. The variable itself actually stores a reference to the area in memory where the object's data and methods are stored:



instantiation

Creating a new object is called *instantiation*. In addition to declaring a variable to refer to the object, the object must be created and initialized in a statement that takes the form:

```
<class> <variable name> = new <class>(<arguments>);
```

The `new` operator allocates memory for the object and returns a reference to the object. `<arguments>` are used to initialize the data for the object.

The code below creates a new object using a class named Circle:

```
Circle spot = new Circle(4);  //spot with radius 4
```

In this statement, the variable `spot` refers to a Circle object that has been initialized with a radius of 4.

To access a member of a class, such as a method, use the object name followed by a dot (.) and then the member name. For example, the code below executes method members getRadius() and area():

```
Circle spot = new Circle(4);
System.out.println("Radius of spot is " + spot.getRadius());
System.out.println("Area of spot is " + spot.area());
```

These statements produce output similar to:

```
Radius of spot is 4.0
Area of spot is 50.24
```

The Circle class will be developed later in this text.

Package Documentation

Package documentation typically summarizes the package classes and the data and methods available in those classes. Code examples that use the classes may also be included in the documentation. Documentation for Java packages are available online at www.sun.com.

Java Packages

Java includes numerous packages as part of the Java Runtime Environment (JRE). These packages contain general use classes, utility classes, or special purpose classes. The most fundamental package is java.lang with classes that define the language itself. Other packages such as java.util have classes for reading input and generating random numbers. These packages will be explained and used throughout this text.

TIP Companies sometimes use their reversed Internet domain in package names. For example, com.mycompany.util names a package with utility classes.

Packages follow a certain naming convention. Java packages start with `java` followed by a dot (.) and then the package name. Companies and other organizations will often name a package with the organization name followed by a dot and then the package name.

The `import` statement is used to make the members of a package accessible to an application. To make a single class from a package accessible, a statement similar to the following is used:

import

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

The class name starts with an uppercase letter, following the appropriate naming convention for class names. If several classes in a package are to be accessible, then a statement that imports the entire package may be used:

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

The asterisk (*) indicates that all members of the util package are to be accessible. `import` statements must appear after a package statement and before any class definitions. Java applications automatically import the entire java.lang package.

java.lang

Obtaining a Value from the User

input stream

An application is more flexible when values can be read from an input stream. An *input stream* is the sequence of characters received from an input device, such as a keyboard. For example, as a user types data, the data goes into the input stream. To process data in the input stream, Java includes the Scanner class with methods for reading integers, floating point numbers, and strings.

Scanner

TIP The Scanner class is new to Java 5.

A program that obtains a value from the user must instantiate a `Scanner` object that is initialized with an input stream. For data typed from a keyboard, initialize the object with `System.in` because it represents the standard input stream.

Scanner class methods include:

Class Scanner (java.util.Scanner)

Methods

`next()`	returns a string from the input stream.
`nextLine()`	returns the string up to the end of line character from the input stream.
`nextInt()`	returns the `int` read from the input stream.
`nextDouble()`	returns the `double` read from the input stream.
`nextBoolean()`	returns the `boolean` read from the input stream.
`close()`	closes the input stream.

The next() method is used for reading a string that does not contain spaces. For example, “apple”. Attempting to read the string “apple pie” generates a run-time exception called InputMismatchException. An *exception* is an error affecting program execution. Exceptions are discussed later in this chapter.

exception

reading numbers and strings

When reading in a combination of numeric and string data, the next() method should be used for reading string data after reading numeric data. Using the nextLine() method is ineffective. It reads the end-of-line character left by the numeric data entry, essentially reading an empty string. If a string with multiple words is expected from the user, the nextLine() can be used to read in the end-of-line character and then another nextLine() used to read the string.

The RectangleArea2 class below instantiates a Scanner object and reads values typed by the user:

```
import java.util.Scanner;

/**
 * Calculates and displays the area of a rectangle
 * based on the width and length entered by the user.
 */
public class RectangleArea2 {

  public static void main(String[] args) {
      int length;     //longer side of rectangle
      int width;      //shorter side of rectangle
      int area;       //calculated area of rectangle
      Scanner input = new Scanner(System.in);

      System.out.print("Enter the length: ");
      length = input.nextInt();
      System.out.print("Enter the width: ");
      width = input.nextInt();
      input.close();

      area = length * width;
      System.out.println("Area of rectangle: " + area);
  }
}
```

Note that the `import` statement appears above the class. RectangleArea2 produces output similar to the following when values `6` and `2` are typed by the user:

```
Enter the length: 6
Enter the width: 2
Area of rectangle: 12
```

In the RectangleArea2 code, a Scanner object is declared, instantiated, and initialized in the statement `Scanner input = new Scanner(System.in)`. The calls to the nextInt() method are used to obtain the values typed by the user. When a call to nextInt() occurs, the application waits until the user types a value and presses enter before the next statement is executed.

prompt

The application also includes *prompts* that tell the user what kind of input is expected. For example, `System.out.print("Enter the length: ")` prompts the user for a length value. When a Scanner object is no longer needed, it should be closed with the close() method.

Review: Distance – part 2 of 2

Modify the Distance application to first prompt the user for the distance of each race segment and then prompt the user for the runner's name. The application should then display the runner's name and total distance to run.

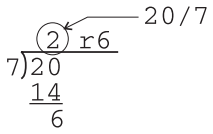
Numeric Expressions

> ***Object Operators***
>
> Objects typically cannot be manipulated with the built-in arithmetic operators. A class usually defines methods that are used to perform operations on objects.

Java includes built-in arithmetic operators for addition (+), subtraction (-), multiplication (*), division (/), and modulus division (%). These operators are for use with primitive data and can be used to form numeric expressions. A *numeric expression* contains at least one operand, which is a value or primitive variable, and may contain operators. For example, the numeric expression `6 + 5` contains two operands and one operator. Numeric expressions can be used in assignment statements, as in the statement `area = length * width`.

Integer division

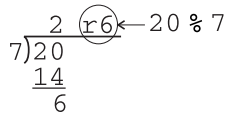
The `/` division operator performs differently depending on the data type of the operands. When both operands are type `int`, the `/` operator performs integer division. *Integer division* truncates the decimal portion of the quotient to result in an integer:



Real division returns the entire quotient, including the decimal portion, and is performed when one or both operators are type `double`.

modulus division

Modulus division returns the remainder resulting from division. The % operator truncates the operands, if necessary, to return an integer:



Modulus division is useful for retrieving digits of a number.

The statements below demonstrate the division operators:

```
int num1 = 5;
int num2 = 3;
int result;
double doubleNum1 = 5;
double doubleNum2 = 3;
double doubleResult;

result = num1 / num2;
System.out.println("num1 / num2: " + result);

doubleResult = doubleNum1 / doubleNum2;
System.out.println("doubleNum1 / doubleNum2: "
                   + doubleResult);

doubleResult = num1 / doubleNum2;
System.out.println("num1 / doubleNum2: " + doubleResult);

result = num1 % num2;
System.out.println("num1 % num2: " + result);

doubleResult = doubleNum1 % doubleNum2;
System.out.println("doubleNum1 % doubleNum2: "
                   + double Result);
```

Executing the statements on the previous page produces the output:

```
num1 / num2: 1
doubleNum1 / doubleNum2: 1.6666666666666667
num1 / doubleNum2: 1.6666666666666667
num1 % num2: 2
doubleNum1 % doubleNum2: 2
```

operator precedence

Java evaluates an expression using a specific order of operations based on operator precedence. *Operator precedence* is the level assigned to an operator. Multiplication and division (including modulus division) have the highest precedence, followed by addition and subtraction. Two operators of the same precedence are evaluated in order from left to right.

order of operations

The order of operations can be changed by including parentheses in a numeric expression. The operations within parentheses are evaluated first. For example, the expression 6 + 4 * 2 – 1 evaluates to 13, and the expression (6 + 4) * (2 – 1) evaluates to 10. It is also considered good programming style to include parentheses when there is any possibility of ambiguity or question about the expression.

Review: Digits

Create a Digits application that prompts the user for a two-digit number and then displays the ones-place and tens-place digits.

Type Casting

Type casting converts a number of one type to a number of a different, but compatible type. For example, a `double` can be cast as an `int`, as in the statements:

```
int i = 2;
double d = 3.7;
int x;
x = i * (int)d     //d is explicitly cast; x is assigned 2x3
```

Type casting is necessary in this case because one of the operands in the expression has less precision than the variable that will store the result. Explicit casting also makes it clear that the programmer intended for the calculation result to be an `int`.

truncate

Casting a `double` to an `int` *truncates*, or removes, the decimal portion of the number, as in the statements above. When decreasing the precision of a number, it is better to round the number. A number with a decimal portion greater than or equal to 0.5 should be rounded up to the next integer, and a number with a decimal portion less than 0.5 should be rounded down. For example, rounding 3.7 results in 4.

TIP Truncation is always toward 0.

rounding

Rounding can be simulated when casting a `double` to an `int` by adding 0.5 to the number before casting, as in the statement:

```
x = i * (int)(d + 0.5)      //x is assigned 2x4
```

If casting a negative number, then 0.5 should be subtracted from the number before casting.

real division

Casting is useful when real division with integers is preferred. The following statements demonstrate the difference between integer division and real division with integers:

```
int i = 5;
int j = 2;
double result;

result = i / j;                    //integer division; result=2
result = (double)i / (double)j;    //real division; result=2.5
```

Java will implicitly type cast operands in a mixed expression to match the precision of the variable storing the result, as in the statements:

```
int i = 2;
double d = 3.7;
double y;

y = i * d      //i is implicitly cast; y is assigned 2.0x3.7
```

Although explicit type casting is not necessary in this case, it is better programming style to include casts. Casting makes a programmer's intentions clear and may make bugs easier to find. Therefore, the statements above should be written as:

```
int i = 2;
double d = 3.7;
double y;

y = (double)i * d      //better code; y is assigned 2.0x3.7
```

Review: GradeAvg – part 1 of 2

Create a GradeAvg application that prompts the user for five grades and then displays the average of the grades. Assume the grades are integer values (for example, 89, 97, and so on). Real division should be performed when calculating the average.

Review: TempConverter

Create a TempConverter application that converts a Fahrenheit temperature to the corresponding Celsius temperature. The formula for converting Fahrenheit to Celsius is C = 5/9(F – 32). The application should prompt the user for the Fahrenheit temperature. Be sure to carefully form the expression. Parentheses will be needed to specify the order of operations.

Formatting Numeric Output

java.text

The NumberFormat class, which is part of the java.text package, is used to create objects that format numbers. NumberFormat objects return a string that contains a formatted number. The formatting of the number depends on which NumberFormat object is used. The NumberFormatExample application creates four different NumberFormat objects:

```
import java.text.NumberFormat;

public class NumberFormatExample {

  public static void main(String[] args) {
    double dollars = 21.5;
    int num = 1234;
    double numWithDecimal = 2.0 / 3.0;
    double sale = .15;
    NumberFormat money = NumberFormat.getCurrencyInstance();
    NumberFormat number = NumberFormat.getIntegerInstance();
    NumberFormat decimal = NumberFormat.getNumberInstance();
    NumberFormat percent = NumberFormat.getPercentInstance();

    System.out.println(money.format(dollars));
    System.out.println(number.format(num));
    System.out.println(decimal.format(numWithDecimal));
    System.out.println(percent.format(sale));
  }
}
```

When executed, the application produces the following output:

```
$21.50
1,234
0.667
15%
```

The DecimalFormat Class

The DecimalFormat class offers additional options for formatting numbers to a specified decimal place. Refer to Java online documentation for more information.

The format() Method

The format() method, discussed in Chapter 3, can also be used to format numeric values. A specifier can take the form:

```
%[alignment][width][.decimal]f
```

where `[.decimal]` indicates the number of decimal places and `f` indicates a floating point number. For an integer, the specifier takes the form:

```
%[alignment][width]d
```

Assignment Operators

In an assignment statement, the expression on the right side of the equal sign (=) is evaluated first and then that value is given to the variable on the left. Because the expression on the right is evaluated before an assignment is made, it is possible to use the current value of the variable in the expression itself. For example:

```
numPlayers = 12;              //numPlayers is assigned 12
numPlayers = numPlayers + 2;  //numPlayers is now 14
```

Changing the value of a variable based on its current value is often done in programming. Therefore, in addition to the = assignment operator, Java recognizes the `+=`, `-=`, `*=`, `/=`, and `%=` assignment operators. These operators perform an operation before making an assignment. For example, the previous assignment statement can be rewritten as:

```
numPlayers += 2;      //numPlayers is now 14
```

The `-=`, `*=`, `/=`, and `%=` operators work similarly, as in the statements:

```
numPlayers -= 3;      //same as: numPlayers = numPlayers - 3
numCopies *= 5;       //same as: numCopies = numCopies * 5
total /= 2;           //same as: total = total / 2
remainder %= 6;       //same as: remainder = remainder % 6
```

Review: GradeAvg – part 2 of 2

Modify the GradeAvg application to use the += operator to sum the grades as they are entered by the user. Format the average grade to display as a percentage.

Using Named Constants

When should a named constant be used?

Named constants should be used wherever they can add clarity to code. However, there are some values that should not be replaced with named constants. For example, in an expression that calculates the area of a triangle (½bh), the value 0.5 is best used instead of a named constant because the value does not have an obvious name.

A *constant* is a name for a memory location that stores a value that cannot be changed from its initial assignment. Constants, like variables, are used in programs so that values can be represented with meaningful names. A constant declaration is a variable declared `final` and takes the form:

```
final <type> <identifier>
```

The declaration begins with `final`, which indicates that the value will not change, followed by the type of data the constant will store and the constant identifier. For example, the following declaration represents π:

```
final double PI = 3.14;
```

`double` declares a numeric value possibly containing a decimal portion. Constant identifiers are typically all uppercase, and may include underscore (_) characters to separate words. For example, `MAX_PRICE`.

The following class uses a constant in the main() method:

```
/**
 * Calculates and displays the area of a circle
 */
public class CircleArea {

  public static void main(String[] args) {
      final double PI = 3.14;
      double radius = 5;      //radius of circle
      double area;

      area = PI * radius * radius;
      System.out.println("Area of circle: " + area);
  }
}
```

CircleArea produces output similar to:

```
Area of circle: 78.5
```

A constant can be assigned a value only once. Trying to change the value of a constant after the initial assignment generates an error. Constant declarations should be grouped at the beginning of a method before any variable declarations.

Identifiers and Keywords

case sensitivity

Identifiers in Java must begin with a letter and may contain letters, numbers, and some special symbols. Periods and spaces are not allowed. Identifiers are also *case sensitive*, which means that an uppercase letter is different from the same letter in lowercase. For example, identifiers `Count` and `count` are viewed by Java as two different identifiers.

The Java language contains *keywords*, which have special meaning to the Java compiler and therefore cannot be used for a variable or constant identifier. The Java keywords are:

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

Although not keywords, `true`, `false`, and `null` are reserved and not for use as identifiers.

Review: CircleCircumference – part 1 of 2

Create a CircleCircumference application that calculates and displays the circumference of a circle. The application should prompt the user for the value of the radius. The circumference of a circle is calculated as $2\pi r$. Use variables and constants as appropriate.

Programming Errors

There are many types of errors that can occur in a program. Some errors are found by the compiler. Others occur at run time. A program that has been carefully designed will have fewer errors. Code reuse can also lead to fewer errors because packages that have been carefully tested and properly documented produce cleaner and more robust code.

syntax error

Errors that violate the rules of Java are called *syntax errors*. For example, forgetting a semicolon at the end of a statement generates a syntax error. Syntax errors are found by the compiler. An application with syntax errors will not run because it will not compile.

logic error, semantic error

A *logic error*, also called a *semantic error*, is more difficult to detect. Logic errors occur in statements that are syntactically correct, but produce undesired or unexpected results, as in the following example:

```
int length;
int area;

length = 3.2;              //3 is actually assigned
area = length * length;    //expected value is 10.24
```

The statements assign the value 9 to `area` rather than the expected 10.24. Although, it is possible that the programmer intended for the value to be truncated, it is more likely that variables `length` and `area` were supposed to be declared as `double`.

Logic errors must be found by the programmer through testing and by carefully examining the source code. Accurate and careful commenting, proper indentation, and descriptive identifiers can help in finding and preventing logic errors.

run-time error, exception

Errors that are not detected by the compiler may generate a *run-time error*. A run-time error, also called an *exception*, halts program execution at the statement that cannot be executed. For example, although the statements below are syntactically correct, they will generate a run-time error because division by 0 is undefined:

```
int totalScores = 40;
int totalTests = 0;
double avgScore;
avgScores = totalScores / totalTests;
```

ArithmeticException

This code generates an ArithmeticException exception.

InputMismatchException

Exceptions can also be generated when user input is not as expected. For example, the application run below generates an InputMismatchException exception because the user typed a string when a numeric was expected:

```
Enter the length: a
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:819)
        at java.util.Scanner.next(Scanner.java:1431)
        at java.util.Scanner.nextInt(Scanner.java:2040)
        at java.util.Scanner.nextInt(Scanner.java:2000)
        at RectangleArea2.main(RectangleArea2.java:12)
```

The program only had code to handle numeric user input. When a letter, rather than a number was typed, an exception was "thrown." Writing code to handle exceptions is discussed in Chapter 12.

Chapter 4 Case Study

This and all subsequent chapters end with a case study. Case studies are used to learn problem-solving techniques. Each case study will include a description, program specification, code design, program implementation, and a discussion about testing and debugging.

In this case study, a Birthday game will be created. The BirthdayGame guesses a player's birthday by having the player perform mathematics with the month and day of their birthday. The number computed by the player is then entered and the program displays the month and day of the player's birthday.

BirthdayGame Specification

BirthdayGame is played with one player. The player is given directions for computing a number that uses the player's birth month and birth day in the calculations:

1. Determine your birth month (January=1, February=2, and so on).
2. Multiply that number by 5.
3. Add 6 to that number.
4. Multiply the number by 4.
5. Add 9 to the number.
6. Multiply that number by 5.
7. Add your birth day to the number (10 if born on the 10th and so on).

BirthdayGame prompts the player for the calculated number and then displays the player's birthday. To determine the player's birthday, 165 is subtracted from the number entered by the player. This number is then divided by 100. The decimal portion of the quotient represents the birth month. The remainder of the division is the birth day.

The BirthdayGame interface should consist of steps that tell the user how to calculate the number that needs to be entered. The player should then be prompted for their number. Finally, the application displays the player's birthday.

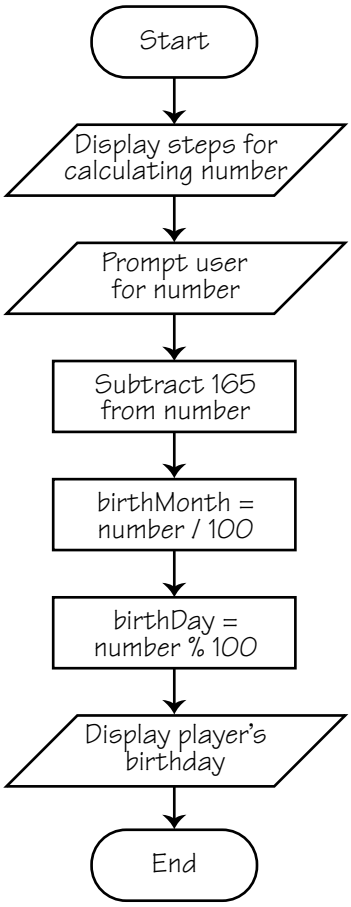
A sketch of a program run:

```
Using paper and pencil, perform the following calculations:

1. Determine your birth month (January=1, February=2 and so on).
2. Multiply that number by 5.
3. Add 6 to that number.
4. Multiply the number by 4.
5. Add 9 to the number.
6. Multiply that number by 5.
7. Add your birth day to the number (10 if the 10th and so on).

Enter your number: 393
Your birthday is 2/28
```

An algorithm in plain English for BirthdayGame:

1. Display the directions for the player to calculate the number.
2. Prompt the player for the calculated number.
3. Subtract 165 from the number.
4. Use integer division to divide the number by 100. Store the quotient as the birth month.
5. Use modulus division to divide the number by 100. Store the remainder as the birth day.
6. Display a message containing the player's birthday.

The BirthdayGame flowchart:



TIP The ▭ flowchart object represents a process.

BirthdayGame Code Design

The code design describes how to accomplish the specification. Included in the code design are a description of the input, output, data generated, and additional flowcharts, algorithms, and pseudocode.

The input for BirthdayGame is a number calculated by the user. An integer variable to store the user's input will be needed.

The output for BirthdayGame is a message with the user's birthday.

Data generated by BirthdayGame is the birth month and the birth day. Integer variables to store the birth month and birth day will be needed.

Based on the algorithm and flowchart, the code design for the BirthdayGame application will include statements for input and output. Calculations will require both integer and modulus division. A pseudocode algorithm for BirthdayGame follows:

```
Display directions (7 steps)
Prompt the user for the calculated number
playerNum -= 165
birthMonth = playerNum / 100
birthDay = playerNum % 100
Display player's birthday
```

BirthdayGame Implementation

Based on the code design, the BirthdayGame implementation follows:

```
/*
 * BirthdayGame.java
 */

 import java.util.Scanner;

 /**
  * Plays a birthday guessing game with one player.
  */
 public class BirthdayGame {

    public static void main(String[] args) {
        int playerNum;
        int birthMonth, birthDay;
        Scanner input = new Scanner(System.in);

        /* Give the player directions for calculating the number */
        System.out.println("Using paper and pencil, perform the following
            calculations:\n");
        System.out.println("1. Determine your birth month (January=1, February=2
            and so on).");
        System.out.println("2. Multiply that number by 5.");
        System.out.println("3. Add 6 to that number.");
        System.out.println("4. Multiply the number by 4.");
        System.out.println("5. Add 9 to the number.");
        System.out.println("6. Multiply that number by 5.");
        System.out.println("7. Add your birth day to the number (10 if the 10th
            and so on).\n");
```

```
        System.out.print("Enter your number:  ");
        playerNum = input.nextInt();
        input.close();

        /* Calculate birth day and month and display result.  */
        playerNum -= 165;
        birthMonth = playerNum / 100;
        birthDay = playerNum % 100;

        System.out.println("Your birthday is " + birthMonth + "/" + birthDay);
    }
}
```

A run of BirthdayGame looks similar to:

```
Using paper and pencil, perform the following calculations:

1. Determine your birth month (January=1, February=2 and so on).
2. Multiply that number by 5.
3. Add 6 to that number.
4. Multiply the number by 4.
5. Add 9 to the number.
6. Multiply that number by 5.
7. Add the birth day to the number (10 if the 10th and so on).

Enter your number: 572
Your birthday is 4/7
```

BirthdayGame Testing and Debugging

This case study performs minimal calculations and the algorithm is simple and straight forward. Simply testing it with your birthday is one method of verifying results.

Chapter Summary

Variables and constants are used in programs so that values can be represented with meaningful names. Variables and constants should be used because they make code easier to read, understand, and modify.

Both variables and constants are created with a declaration statement. A variable declaration includes the data type and identifier. A constant declaration also includes the keyword `final`. Identifiers are case sensitive and cannot be the same as a Java keyword. The value of a variable can be changed throughout program execution with assignment statements. The value of a constant cannot be changed from its initial assignment.

A primitive data type stores a single piece of data and can include `int`, `double`, `char`, and `boolean`. Abstract data types include classes. Each class defines not just a single piece of data like a primitive data type, but a set of data along with methods for performing actions on that data. Variables declared with an abstract data type are called objects. An object declaration is called instantiation. An object is instantiated with the keyword `new`.

Data can be read from the user at run time by using the Scanner class. This class processes data from the input stream. Objects that read data from the keyboard are initialized with `System.in`. When obtaining data from the user, a prompt should be included so that the user knows what information is expected.

Java includes many packages. Package members are accessible with an `import` statement. The Scanner class is in the java.util package. The NumberFormat class is in the java.text class. The NumberFormat class is used for formatting numbers.

Java includes built-in operators that are used to form numeric expressions. Arithmetic operators include `+`, `-`, `*`, `/`, and `%`. The / operator performs integer division when both operands are type `int`, and real division when at least one operand is a `double`. Modulus division is performed with the % operator. An expression is evaluated according to operator precedence. The order of operations can be changed with parentheses. Assignment operators include `+=`, `-=`, `*=`, `/=`, and `%=`. Each of these operators perform an operation before making an assignment.

Type casting converts a number of one type to a number of a different type. Casting is useful when real division with integers is preferred. A `double` can be rounded by first adding 0.5 before casting to an `int`.

Programming errors occur for many reasons. A syntax error violates the rules of Java. A logic error is also called a semantic error and is more difficult to detect because statements are syntactically correct, but produce unexpected results. A run-time error is also called an exception. An exception halts program execution.

Code conventions introduced in this chapter are:

- Variable identifiers begin with a lowercase letter and any word after the first within the identifier should begin with an uppercase letter.
- Constant identifiers are all uppercase. Multiple words in an identifier can be separated with an underscore (_) character.
- Variable and constant declarations should be grouped at the beginning of a method.
- Each line of a program should contain only one statement.