COP 2250 – Introduction to Java Programming

**Chapter 1**

The Java content starts on in Section 1.5 on page 10. Start there and continue through to the end of the chapter, but ignore Section 1.11. You will be using Eclipse, not NetBeans.

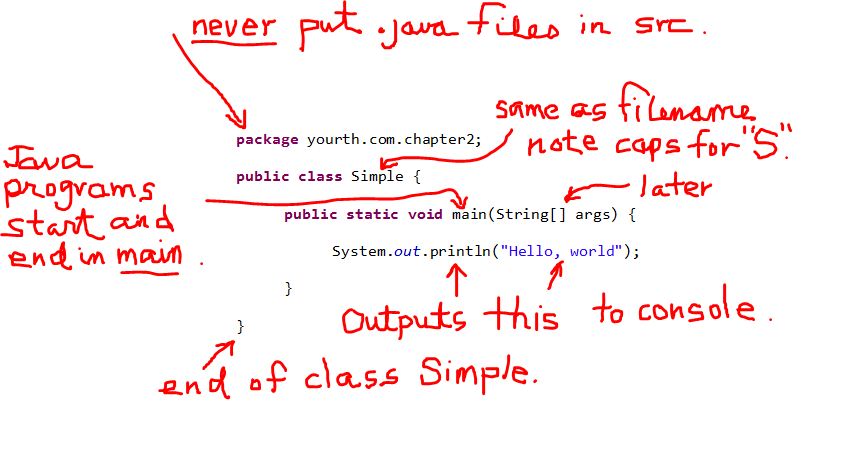
* Java was created by a team led by **James Gosling** at Sun Microsystems (now in Oracle).
* Java programs compile into **byte code**, not machine language. To run a Java program, a computer needs a Java Virtual Machine (**JVM**) that can interpret the byte code.
* A Java program written on a Mac can run on a PC or Linux computer if it has a JVM.

Some Key Points

* **Comments** start with //, on a new line or at the end of a line.
* Java is **case-sensitive**.
* Java statements end with a **semicolon**.
* To start a new Java program, you pick a name for a **class** (1st character in upper case).
* If a class name consists of several words, capitalize the first character of each word.
* This class name must be identical to the name of the **.java** source code file.
* Only one class can be **public** (means access is unrestricted) in a .java file.
* For now, all Java programs will have a **main method**. This makes the program executable. Later, we will create classes to model our data. They won’t have a main.

Section 1.7

* This section covers the parts of a basic Java program with several examples.
* NOTE: the package statement must be at the top of every program file.



Section 1.8

* This section explains how Java source code is compiled into byte code.
* Figure 1.6 on page 16 depicts this well.

Section 1.9

* Programming style and documentation are important.
* Note well the indentation and block styles on page 19. Use the end-of-line style.
* Eclipse (covered below) can quickly help you to create nicely styled, and easily to read, source code files. Use Source -> Format and Source -> Correct Indentation.

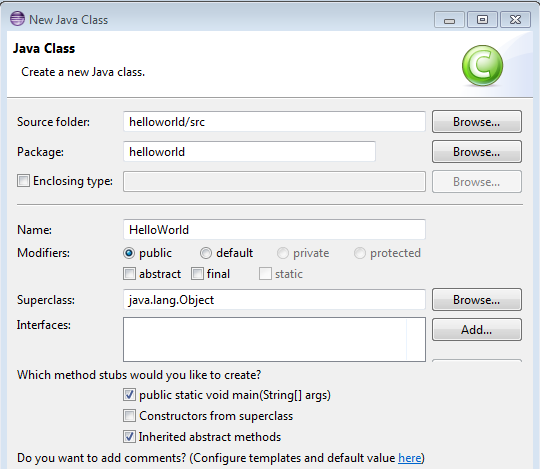
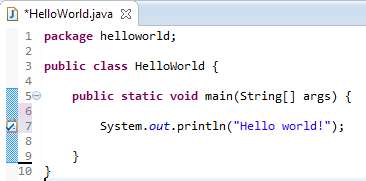
Section 1.10

* Note the three types of programming errors.
* Eclipse will detect and identify your syntax errors and even suggest fixes.

Section 1.12 - Eclipse (most used Java IDE)

* When you start Eclipse, you will be prompted to specify a “**workspace**”. This is just a folder where your Java will be saved. I suggest you create a folder on your desktop with your name on it, like smithjava. Use this workspace throughout the course.
* When the IDE appears, it will be in the **Java perspective,** a set of window panels best for Java. Eclipse can be used for other programming languages.
* Eclipse programs are always part of a **project** that can contain many programs.

Starting a new Eclipse Project and Program

1. Select **File > New > Java Project**
2. This opens a “New Java Project” dialog window.
3. Enter the project name in the box at the top (all lower case). Change nothing else, then click Finish, or Next > Finish if you like mouse-clicking. ☺
4. The project will appear in the **Package Explorer** window on the left. Expand it.
5. Don’t open or mess with the JRE System Library.
6. Right-click on the project name and select **New > Package** from the popup. You can alternatively click the **package icon**  on the toolbar. That will suggest a package name that is identical to the project name, and that is okay.
7. Don’t create your Java programs in the default **src** folder as is shown in the book. Always create a package. A package is a folder in your project.
8. To create a program, right-click on the package icon in the Package Explorer window and then select New > Class. You can also click the class icon  on the toolbar.
9. The “New Java Class” dialog will open. Pick a class name (HelloWorld) and check the public static void main checkbox as shown. Click Finish.
10. By convention, Java class names, and the first character of each word in multiple word class names, begin with an **upper case character**. This is important.
11. Eclipse will then display a skeletal source code file named **HelloWorld.java** in the large editor window.
12. Start Java coding inside the main method as shown here.
13. To run a program in Eclipse, click the arrow icon  on the toolbar.

The print and println Methods

* These are your console output methods. In Eclipse, they generate output in the Output window below the editor window in Eclipse.
* The println() method outputs its argument and causes a new line.
* The print() method does not make a new line, so subsequent output is on the same line.

**Chapter 2**

Keyboard Input from Console

* The **Scanner** class can be used for keyboard input and interactive Java programs.
* Scanner is not in the default **java.lang** package so you have to **import** it from the **java.util** package.
* All import statements follow the package statement and precede the public class header line.
* The keyboard in Java is the **in property** of the System class, **System.in.**
* To use the Scanner class, you create an instance of Scanner and assign its address to System.in.

Scanner keyboard = new Scanner(System.in);

* Eclipse will generate a warning for the line above. To eliminate the warning, enter the following statement when done with inputs:

keyboard.close();

* Once the Scanner object has been instantiated, it can be used to run any one of many methods of the Scanner class.
* Commonly used methods for numeric input are shown in Table 2-2 on page 45. The table shows that each numeric data type has a corresponding Scanner method.

Identifiers

* These are names that you select to refer to your data.
* Always choose identifiers that…um…***identify*** what they are storing in memory.
* For example, weekly**P**ay is a better choice than x for a double type that would store a person’s weekly pay. Note the “camel case” style of weeklyPay is common in Java.
* There are strict rules for choosing your identifiers in Java. See the bullet list and notes on page 40.

Class Names

* By convention, class names start with an upper case character.
* Multiple word class names use upper case for the 1st character in each word.

Some good and not good class names:

AcmeEmployee good

MonthlyRateOfInterest good

Myfirstprogram not good use MyFirstProgram

Variables and Literals

* Java is a strongly typed language. That means that you must declare the type of data that you wish to store in the variable when you are creating the variable in code.
* An integer like 10 or -6 is stored in the **int** type. A number with decimals is stored by default as a **double** type.
* Variables must be declared before they are used. The value in the variable can be optionally assigned when it is declared.

Constants with final

* The final keyword is used to assign a constant.
* The type and value of the constant must be specified when creating a constant.
* By convention, constants in Java are written in all **upper case, with underscores between each word** of a multi-word name.

final double STATE\_SALES\_TAX = 0.06;

final int MAX\_SIZE\_OF\_CLASS = 24;

Numeric (Primitive) Data Types

* In Java, every piece of data you use in a program has a data type.
* Classes are **object types**.
* Numbers, however, are usually stored as non-object **primitive types**.
* See Table 2-1 on page 45 for a list of common Java primitive types for numbers.
* Note also the amount of memory each type needs and the range of possible values that each type can store.
* Since they are the defaults, we will use type **int** for integers and type **double** for floating point numbers, but if memory space is a concern, coders always use the most suitable type.
* Note the Scanner class methods for numeric input in Table 2.2 on page 45.

Arithmetic Operators

* See table 2-3 on page 46.
* Note that when an integer is divided, any decimal part in the quotient is dropped.
* NOTE: when using the **/** operator with int types, any decimal fraction part is lost.

int num = 25 / 7; // stores 3 in the variable num.

* NOTE: the **%** operator (remainder or **modulus**) yields the remainder after a division.

int num = 25 % 7; // stores 4 in num.

The Math Class

* This class is always available to you. It’s in the default java.lang package.
* The Math class has many useful methods that you can employ.
* Math.pow() is used for exponentiation, and Math.sqrt() for finding square roots.

Floating Point Types

* Note that, as mentioned previously, floating point values are doubles by default.
* If the literal is followed by **F or f**, a float type can be stored.

Operator Precedence

* This is just the same as in math class. “Please Excuse My Dear Aunt Sally” is in effect.

The Combined (Augmented) Assignment Operators

See Table 2-4 on page 54.

Increment and Decrement Operators

* See Table 2-5 and examples on page 55.

Conversion Between Numeric Data Types

* Examine the pages 56-58.
* The primitive data types are ranked on byte size.
* Java will convert a lower ranked type to a higher ranked type without a problem.
* But a higher ranked type cannot be automagically converted to a lower ranked type. In a case like this, a **cast** is required.

Primitive Types and Class (Reference) Types

* The value in a variable of primitive type is the actual value assigned to it.
* The value stored in a variable of an object type is a memory address. The variable doesn’t store the object, it stores a reference to the object. Objects are also called “reference types” for this reason.

Common Errors to Avoid

* See pages 65-67.

COP 2250 – Java Programming 1 – Chapter 3 - Selections

Boolean Operations

* Recall that a Boolean variable can have only one of two values: true or false.
* Choose self-documenting names for Booleans like “isTeenager” or “hasCash”.
* Booleans are often used with operators to create expressions.

Comparison (aka Relational) Operators (page 76)

* These operators test a relationship between two values and return a Boolean.

|  |  |
| --- | --- |
| **Symbol** | **Meaning** |
| < | less than |
| > | greater than |
| <= | less than or equal to |
| >= | greater than or equal to |
| == | equal to |
| != | not equal to |

if Statements (page 78-87)

* If statements enable Java programs to branch and execute along alternate paths.
* These use the operators above to create Boolean expressions.

Simple ifs do not use an else block.

if(Boolean expression) {

// statement(s) to run if expression is true

}

* Try SimpleIfDemo on the textbook’s web site.

Two-way if-else blocks consist of both if and else blocks.

if(Boolean expression) {

// statement(s) to run if expression is true

}

else {

}

// statements to run if expression is false

Nested ifs and Mult-Way if-else Statements

* else if blocks are used for more than two possibilities.
* if and/or else blocks can be nested within other ifs to any depth desired.
* using good indenting form will promote readability.
* NOTE the preferred format for multiple alternative ifs on page 82.

Common Errors and Pitfalls

* See page 83-87

|  |  |
| --- | --- |
| **Avoid This** | **Do This Instead** |
| **if(num == true) {**  **}** | **if(num) {**  **}** |
| **if(num == false) {**  **}** | **if(!num) {**  **}** |

Generating Random Numbers (See page 87)

* These can be generated with the random( ) method of the Math class.
* To generate a random number between 0 and 100, use:

int number = (int)(Math.random( ) \* 101);

* To generate a random number between 1 and 100, use:

int number = (int)(Math.random( ) \* 100) + 1;

* Try SubtractionQuiz on the textbook’s web site.
* Try ComputeAndInterpretBMI on the textbook’s web site.
* Try ComputeTax on the textbook’s web site.

System.exit(0);

* This statement will immediately stop an executing Java program (see page 92).

Boolean (Logical) Operators (page 93)

* These connect two Boolean values to create a new Boolean value.

|  |  |
| --- | --- |
| **Symbol** | **Meaning** |
| ! | logical NOT |
| && | logical AND |
| || | logical OR |
| ^ | logical XOR |

* Study the truth tables on pages 93-94.
* Use && when both expressions must be true
* Use || when either expression can be true
* Use ^ when either expression can be true, but not both
* && performs a short-circuit test. If the left side of the && evaluates as false, the other side is not evaluated (since the entire expression will be false)
* NOTE: if condition is compound, each part must be capable of standing alone.

|  |  |
| --- | --- |
| **Incorrect** | **Correct** |
| **if(num == 4 || 8)** | **if(num == 4 || num == 8)** |

* Try LeapYear on the textbook’s web site
* Try Lottery on the textbook’s web site

Exercise

Write a program called Teenager.java that prompts the user to enter his/her age (as an integer) and then reports whether the user is a teenager or not. Use a compound expression as the if condition.

Exercise

To vote in the U.S.A, a person must be a citizen and at least 18 years old. Write a Java program called Voting.java that prompts the user for his/her age, asks the citizen question, and reports on voting eligibility. Use nested ifs.

The switch statement (See page 100)

* This control structure often offers better coding than a series of if … else if blocks when your program requires multiple branches. The general format is:

switch ( expression ) {

case constant 1:

statement(s)

break;

case constant 2:

statement(s)

break;

case constant 3:

statements(s)

break;

…

default:

statement(s) to run if no case above is matched

} // end of switch

* note the colon after each switch case label.
* the break statements are required, otherwise code in following cases is executed, too.
* the default statements are executed if no cases are a match.
* NOTE: switch case labels can be stacked, if desired. See page 101.

Conditional Expressions (See page 103)

* This amazingly terse operator is an alternative to an if…else set.
* The syntax is:

(Boolean expression) ? true statement : false statement

* Example:

int numberOne = 8;

System.out.println((numberOne < 10) ? “Less than 10” : “10 or more”);

* The output from the statements above is: Less than 10

Exercise

Write a program named OddEven.java that prompts the user to enter an integer, and then reports whether the entry is odd or even. Use a conditional expression to generate the odd or even response.

Operator Precedence (See page 104)

* NOTE Table 3.8 on page 105.

Debugging

* Note the bullet list on page 106.
* See also the “Eclipse Debugging” video in D2L.

COP 2250 – Java Programming 1 – Chapter 4 – Math, Characters, & Strings

The Math Class

* This class is in the java.lang package and so is always available in your programs.
* There are many methods and a few constants. See pages 120-124.
* NOTE that the trig methods measure angles in radians, not degrees.
* The Math.random() method is an important method for generating random numbers.
* With casting to (int), random integers can be created. See page 122.

The Random Class

* This class in the java.util package can also generate random numbers.
* It has methods for creating random numbers in various numeric data types. See page 335.
* This class is tested on the 1Z0-811 Exam.

Try ComputeAngles

The Character Data Type

* This data type stores a single character, using the **char** keyword.
* Use single quotes (apostrophes) when assigning a char.

char ch = ‘A’;

* Characters can be encoded using ASCII or Unicode.
* ASCII is an 8-bit scheme with 256 possible characters. See Table 4.4 and Appendix B.
* Unicode is a 16-bit scheme so 65,536 characters were possible.
* Even that could not handle all characters in all languages so Unicode was extended.
* You can use either system:

char ch = ‘a’; // ASCII way

char ch = ‘\u0061’; // Unicode equivalent (recall that hex 61 = 97)

* char is actually a numeric type. See Note on page 126.

Escape Sequences

* See page 126 and Table 4.5

Casting between char and Numeric Types

* This is possible because char is a numeric type.
* See page 127 for examples.
* Implicit casting without ( ) works if the cast fits into the target type, otherwise explicit casting with ( ) is required.
* Numeric operators can be applied to char variables.

Testing Characters

* You could use if expressions with the ASCII range for the sought characters as shown just above Table 4.6 on page 128.
* However, Table 4.6 shows the Character class methods that make this task much easier.

The String Type

* String is a defined class in the Java API.
* String variables are reference, (object) types.
* The value stored in a string variable is the address of the string in memory.
* In contrast, numeric types like int, double etc. are primitive types. The data they store is the assigned numeric value.
* Table 4.7 shows some simple methods of the String class. There are many more in the API.
* Study well the String skills and examples on pages 130-138. You will need these skills often.
* NOTE well the difference between using the **next()** and **nextLine()** methods of Scanner for inputting strings. The Caution in the middle of page 133 covers the danger.
* Table 4.8 shows the String class comparison methods.
* Note well the Caution for comparing strings with relational operators on page 134. These operators would compare the addresses of the strings, not the strings themselves.

Try OrderTwoCities

* Tables 4.9 and 4.10 show the String class methods used for finding substrings.

Try GuessBirthday

Try HexDigit2Dec

Try LotteryUsingStrings

Formatting Console Output

* Control of decimal places in numeric output can be accomplished with System.out.**printf()**.
* General format is:

System.out.printf(“formatting string”, item(s) to format);

* The formatting string contains **format specifiers**, one for each item to be formatted.
* Table 4.1 on page 146 displays the commonly used format specifiers.
* The items must match the type and sequence of the format specifiers in the string.
* Note well the example at the very bottom of page 146.
* Table 4.12 on page 147 displays specifiers for width and precision.
* Note the important Caution and Tip on page 148.

Try FormatDemo

COP 2250 – Java Programming I - Chapter 5 - Loops

# Loops

* Many computer programs require a block of code to be executed more than once.
* This can be done with loops.
* Java has three looping statements. All of them evaluate a Boolean expression to determine if the loop should perform another **iteration**, or cycle.
  1. while
  2. do-while
  3. for

# The while Loop

* A while loop will cycle as long as a conditional expression inside ( ) evaluates as true.
* Note that this is a pretest loop. If the condition is initially false, the entire loop is skipped.
* General form:

while( Boolean expression )

// **single** statement when expression is true

But it is recommended that you enclose even a single statement in a { } block as below:

while ( Boolean expression ) {

// statements **block** to repeat when expression is true

}

* Care must be taken to ensure that the Boolean expression will eventually test as false, else an **endless**, or infinite, loop results.

Try RepeatAdditionQuiz

Try GuessNumberOneTime

Try GuessNumber

Try SubtractionQuizLoop

# Exercise

Write a program to print the multiples of 7 between 90 and 60. Use a while loop and progress down from 90 to 60. Don’t delete this when done, we will add to it in Exercise 3.

# Counting Loops

* In programming, a counter is a variable that is **initialized to zero and incremented** every time that a certain condition is true.
* Integer types are always used for counter variables.
* Counters can be used to count the number of iterations that a loop runs through.
* A counter can be used with if to cause a block of code to execute based on the condition.

# Exercise 3

Modify Exercise 2. Display the multiples of 7 between 90 and 60 as before, but add a counter to count the number of multiples. Report the count after the loop ends.

Sentinel-Controlled Loops

* A while loop condition can be set to watch for a special value, called a **sentinel**, from input.
* Typically, the sentinel is used to stop the loop.

Try SentinelValue

# Exercise 4

Write an interactive program that asks the user to enter a number, and then displays the square root of the number. The program should continue until the user enters zero or a negative number. If the user does, the program should terminate. Use a sentinel.

**do-while Loops**

* The general format is: **do {**

**//statements to repeat**

**} while (test condition);**

* Note that this is a posttest loop. The test condition is at the end of the construct.
* This means that a do-while loop **always executes at least once**.
* Note also that it **ends with a semicolon**.

Try TestDoWhile

### **Exercise**

Write a Java program that uses a do-while loop to display the numbers from 1 – 10, all on one line of output.

# The for Statement

* The general syntax is:

**for(initialization expression; test condition; modifying expression)**

**// single statement to repeat**

* **Only the very next line** is repeated, but a block inside braces is recommended.
* To repeat a block of statements, enclose the block within { }.
* Many Java shops will demand this as a coding convention.

**for(initialization expression; test condition; modifying expression){**

**// multiple statements to repeat**

**}**

# Omitting Expressions in for Statements

* Any/all of the for loop expressions in the first line can be omitted, but there will always be two semicolons in this line.
* An infinite for loop can be created with:

**for ( ; ; ) {**

**// statements to repeat**

**}**

* Java allows you to declare the initialization variable inside a for statement.
* Instead of:

**int i;**

**for(i = 0; i < myNums.length; i++) {**

… you can use:

**for(int i = 0; i < myNums.length; i++) {**

* This latter form is recommended unless you need the variable later in your code (since it is not visible outside of the loop).

# Commas In for Statements

* Both the initialization and modifying expression may have commas.
* This can be done when a loop has **several counters** to initialize and update.
* Each expression must be capable of standing alone.

# Exercise

# Try this code. Copy-paste it into a class named LoopFor in Eclipse. Edit the package statement.

// LoopFor.java

package yourpackage;

public class LoopFor {

public static void main (String [] args) {

//int j;

for ( int i = 0, j = 10; i <= 10; i++, j-- ) {

System.out.println ( "i = " + i + ", j = " + j );

}

}

}

Which Loop to Use?

* See page 174-175.

### **Nested Loops**

* Nesting occurs when a loop contains other loop(s).
* Nested loops are useful for displaying **tabular** data.
* Try MultiplicationTable

### **Exercise**

Use nested for loops to generate this output. Note: **\t** is the tab escape code.

1.0 1.1 1.2 1.3 1.4 1.5

2.0 2.1 2.2 2.3 2.4 2.5

3.0 3.1 3.2 3.3 3.4 3.5

4.0 4.1 4.2 4.3 4.4 4.5

5.0 5.1 5.2 5.3 5.4 5.5

**Exercise**

Write a Java program that uses an endless loop to continuously dump Pythagorean triplets (with no number being greater than 100) to the screen. Repeats and multiples are okay. Create random numbers with the Math class. Pythagorean triplets are values that can make a right triangle, such as 3,4,5 and 5,12,13 etc. You can stop the endless loop in Eclipse by clicking the tiny red square icon.

**Minimizing Numeric Errors**

* Using floating-point types to control loops can cause approximation errors

Try TestSum

Try GreatestCommonDivisor

Try FutureTuition

Try Dec2Hex

Keywords break and continue

# The break Statement

* This statement will immediately **halt** an executing loop.
* Use it to prematurely break out of a loop when some condition becomes true inside the loop.
* Sometimes you will want several exit points for a loop.
* A selection statement is commonly used to conditionally execute the break statement.

### **The continue Statement**

* This statement causes a loop to **skip** to the end of the loop body.
* If the loop test condition is still true, the loop starts the next iteration.

Try TestBreak. Try TestContinue. Try Palindrome, Try PrimeNumber

**COP 2250 Java Programming I - Chapter 6 – Methods**

**What a Method is**

* A method is a named block of code with braces that enclose one or more Java statements.
* Methods organize and simplify programs and make code reusable.
* There are many methods in the Java API. Each one is defined in a Java class.
* However, you can also craft your own methods. That is the subject of this chapter.
* View Figure 6.1 on page 205. It illustrates the terminology for the parts of a method.
* Note: All methods in this chapter will have the **static** modifier.
* Some methods **return a value** as indicated by the return value type.
* Note that a method definition has:
  + a method **header** line with:
    - two modifiers, public and static
    - a return value type
    - a name
    - parentheses that optionally may contain a list of parameters
  + a **body** consisting of statements inside curly braces
* The value returned by a method should be “caught” so it can become part of the program.
* Methods that return a value always have a **return statement** to do the job.
* Some methods do not return a value. These methods are always of return-type **void**.
* Some methods require one or more **parameters** inside parentheses when they are defined.
* Parameters are values that the method needs to use internally to do its job.
* Methods can be called (aka invoked or executed) by other methods. For now we do this in main.
* When a method is called a value called an **argument** is passed to each parameter. This is explained well by Figure 6.2 on page 207.

Try TestMax

Try TestVoidMethod

Try TestReturnGradeMethod

* Although a void method doesn’t require a return statement, a naked return statement can be used to halt a void method. See page 210.

**Exercise**

Write a program named Adding.java that has a method named addTwo(). The addTwo( ) method should take two **double** parameters and return their sum. Test the addTwo() method with several calls in the main method. Don’t forget the static modifier in the method header and be sure to output the sum in main.

**Passing Arguments by Values**

* In the definition of a method, the parameters must be **typed**.
* When the method is called, values passed to the method are known as **arguments**.
* The arguments must match the parameters in **order**, **number** and **data type.**
* A parameter is a variable with its own location in memory, and when a method finishes, the parameter is cleared from memory.
* Parameters are said to be **passed-by-value** to the method. See page 212.

Try Increment

Try TestPassByValue

**Modularizing Code**

* Programs can often be improved by organizing statements into methods.
* This modularization also promotes **reuse** and eliminates **redundancy**.
* Using loops to call one or more methods in every loop cycle can greatly simplify your code.

Try GreatestCommonDivisorMethod

Try PrimeNumberMethod

Try Hex2Dec

**Overloading Methods**

* Overloading means that one class can have more than one method with the **same name**.
* Each overloaded method will have its own body statements.
* Overloading is permissible only if:
  + The **number** of parameters is different.
  + The **data types** of the parameters are different.
* NOTE: Using a different return type or modifier is NOT permitted for overloading a method because of the ambiguity. See page 221.

Try TestMethodOverloading

**Exercise**

Create a new Java class called **Addem.java** that has a method named **addNumbers( )**. The

addNumbers( ) function should take 2 integers as parameters and return the **sum** of the 2 numbers. This method should also report (to the screen) the two numbers being added. Catch and display the value returned by this function. Test it with some low numbers so you can easily check the result. When done, overload addNumbers( ) to accept **doubles** and **bytes**. Test both new methods.

**Variable Scope**

* The visibility of a variable in a program is called the **scope** of the variable.
* A variable declared inside a method is visible (accessible) only inside that same method. It has **method scope.** A parameter passed into a method also has method scope.
* A variable cannot be used until it has been declared (by name and type) and assigned a value.
* See Figure 6.6 on page 223.

Generating Random Characters (see page 223)

* Recall that char is a numeric type, so random characters can be generated with Math.random().

Study RandomCharacter on page 224. Then try TestRandomCharacter

The chapter ends with a look at abstraction and stepwise refinement.

Try PrintCalendar

COP 2250 - Chapters 7 and 11 – Arrays & ArrayLists

Chapter 7 – Single Dimension Arrays

# NOTE: Chapter 8 will be covered in COP 2251

# Array Basics

* An array is a collection of variables of the same type with one name.
* Put another way, an array is a variable with multiple “slots” for values of the same type.
* The variables in an array are called the elements of the array.
* The elements are indexed numerically in sequence, starting from zero.
* The elements of an array can be of any type, including objects.
* However, in Java, all elements of a particular array must be of the same type.
* Arrays are immutable. They cannot be re-sized. Element values can be re-assigned but elements cannot be deleted, inserted, or added.
* Arrays are declared with [ ] , either before or after (more common) the array name.

double[] prices; // declares array of doubles prices

String[] friends; // declares array of Strings friends

int nums[]; // adopted to please C programmers tsk tsk

* The new keyword is used to allocate space for the array.

prices = new double [25]; // makes space for 25 doubles

nums = new int [12]; // makes space for 12 ints

You can’t use an array until you allocate space for it

* Declaration and space allocation can be combined in one step, if desired:

String friends[] = new String [30]; // C syntax

double[] monthlySales = new double [12];

* Default values are assigned to array elements when space is allocated. Numbers are assigned zero, Booleans are assigned false, objects are set to null, and chars to ‘\u0000’.
* If the values to be stored in an array are known, the array can be declared and initialized with these values when the array is created. Then, new is not needed. Note curly **braces**.

int[] carsSoldThisWeek = {2,5,0,3,2,2,1}; //**array initializer**

# Array Subscripting (indexing)

* The elements of an array are indexed, starting from zero. See Figure 7.1 on page 247.
* The seven element carsSoldThisWeek array above is indexed from 0 - 6 as shown next :

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 2 | 5 | 0 | 3 | 2 | 2 | 1 |
| indexes | 0 | 1 | 2 | 3 | 4 | 5 | 6 |

* Individual array elements are accessed by specifying the index inside [ ]

System.out.println ( carsSoldThisWeek[2] ); // output is 0

* The value stored in an array element can be reassigned in a similar manner:

monthlySales [3] = 5432.88;

* When the array elements are objects, their methods can be executed:

String[] friends = new String [30];

friends[4] = “Bud Mack”;

System.out.print(friends[4].charAt(4)); // output is M

* Every array has a length property that holds the number of elements in the array.
* The index of the last element in an array is one less than the length of the array.

# Processing Arrays

* Arrays can be easily processed with loops, either while loops or for loops.
* The key is to use the loop control variable inside the [ ] to cycle through an array.
* Note well the nine specific examples commonly required for processing arrays on p 249-251.
* Java also has an enhanced for loop (foreach loop), but it’s one-way only (ascending). See page 251. With it, you can “traverse” an array without using indexes.

int[] numbers = {12,3,34,15,9,8, 24};

for (int n : numbers) {

System.out.print(n + " ");

}

Output is 12 3 34 15 9 8 24

* Note the Caution on page 251. Attempting to access the array element at a non-existent index throws an **ArrayIndexOutOfBoundsException**, crashing the program.

Try AnalyzeNumbers

Challenge Exercise

Write a program that declares a five element array of doubles. In a loop, prompt the user to enter five test scores (percent scores) and assign them to the array. Use the length property of the array. Use a second loop to display the scores again. Use an accumulator to sum up the scores in the second loop. An accumulator is a variable that is initialized to zero and subsequently used as a “bucket” to accumulate other values. End the program by displaying the average percent score.

# Accessing Array Elements Sequentially

* there are many occasions when a Java program will need to access an array sequentially:
  + To search for a particular element.
  + To count the number of occurrences of the same value.
  + To determine the largest and/or smallest element.

# Challenge Exercise

Write a program that prompts the user to enter five integers from the keyboard. Store the integers in an array. Process the array to determine both the smallest and largest integer. Report the results.

Deck of Cards Study

* This is an interesting and instructive study on page 254.

Try DeckOfCards

Copying Arrays (See p 256)

* A simple assignment like array2 = array1; doesn’t copy array elements. It copies the reference (address) of array1 to array2, making array1 and array2 reference the same array. This happens because an array is an object type in Java.
* Use a loop or the **arraycopy( )** method of the **System** class to copy an array. See p 257.
* NOTE: that for arraycopy, the target array must have been declared and assigned space.

Passing Arrays to Methods (p 257)

* A method can receive either a named array or an anonymous array as a parameter. See p 258.
* The method receives a reference to, not a copy of, the array argument. This means that the method will have access to where the array lives in memory (and can change the array element values).
* This is called **pass-by-sharing** by Liang.

Returning an Array from a Method

* A method can return an array. The returned array must be “caught” in an array variable of the correct type or used as the argument to another method.

Try CountLettersInArray

Variable-Length Argument Lists (for a method)

* A variable number of arguments of the same type can be passed to method with the syntax **type… paramName**.
* Only **one** variable length argument is permitted, and it must be the **final** one in the list.
* Inside the method, Java creates an array for the variable length parameter.

Try VarArgsDemo

Searching Arrays

* A common programming task is to search an array for a specific value. Liang provides sample code for two algorithms for searching. See page 216.
  1. Linear Search
* Searches the array from start to finish until a **match** is found for the **key**.
* This inefficient approach can be time-consuming for a large array.
  1. Binary Search
* Assumes an **ordered** array and starts search with a **comparison** to the **middle** element.
* Each successive trial eliminates half of the remaining elements, a more efficient approach.

Sorting Arrays

* This is another common task in programs that use arrays.

Selection Sort

* To sort in ascending order, this method repeatedly selects the largest value and swaps it with the last value in the remaining values.
* See page 269 and watch the **Video Note**.

Try SelectionSort

The Arrays Class

* This utility class is in the **java.util** package and must be imported to your programs.
* The class has static methods that simplify some array operations, including sorting.
* See page 270-272.

Command Line Arguments

* It’s time to finally learn what the **String args[]** thing is all about in main.
* The **args** array is used to read strings passed into main from the command line.
* Command line args can be easily emulated in Eclipse for convenience.
* To add arguments, select the desired main class in the Package Explorer and then, on the Eclipse menu, select **Run | Run Configurations…**
* In the resulting dialog box, click the **Arguments** tab.
* Enter the desired arguments in the textbox.
* There is a short video for this, too. Watch it back in MyCourses.
* Test your Eclipse command line arguments skill with the **Calculator** program below.

Try Calculator

Class Arraylist

This important class is tested by the 1Z0-811 exam. ArrayList is covered in two locations in our text:

* Chapter 11: page 432 – 440
* Chapter 20: page 766 – 771
  + The important part of this section is the use of iterators with ArrayList.

This API class solves the issue that an array cannot be re-sized. An ArrayList is a data structure with methods for adding, deleting, inserting, and manipulating elements. See the UML diagram on p433.

* However, an ArrayList cannot store primitives like int or double. It’s for storing objects of the same type, not primitives. That doesn’t mean that you cannot store numeric data in an ArrayList. You just have to use a wrapper type like Integer or Double to store numbers.
* The object type to be stored in an ArrayList is specified inside < > when it is declared.

ArrayList<Integer> amounts = new ArrayList<>(); // array list of Integers

ArrayList<Double> prices = new ArrayList<>(); // array list of Doubles

ArrayList<String> names = new ArrayList<>(); // array list of Strings

Although an ArrayList cannot hold primitives, autoboxing and autounboxing is allowed to simplify the syntax. See pages 383 – 384.

amounts.add(new Integer(15)); // adding an element with the wrapper class

amounts.add(33); // adding an element with autoboxing

Try TestArrayList

* Table 11.1 on page 436 compares and contrasts arrays and ArrayLists. Study it well.

Try DistinctNumbers

* Like an array, an ArrayList can be processed with an enhanced for (foreach) loop. See the code snippets in the middle of page 437.

Useful Methods for Lists

* Refer to page 438-439 for some important skills for ArrayLists.

Chapter 20

* Read pages 766-771 to learn about iterators and their usage with ArrayList.

Try TestArrayAndLinkedList.java from the zip (but ignore the LinkedList code).

Try ArrayListOne from the zip.

COP 2250 – Java Programming I - Chapter 9 - Objects and Classes

# Objects as Models

Java is an object-oriented language. Java programs can model the real world, using objects in code to represent actual physical objects or intangible objects. A Java programmer working for AmTrack would create and work with boxcar objects. A Java programmer working for Coca Cola would use objects for bottles, cans, cases, and delivery trucks.

Objects:

1. Have state

* This means objects contain one or more items of data with assigned values.
* These items of data are called attributes, fields, instance variables, or properties.

1. Have behaviors

* This means that objects respond to events to perform operations.
* These operations are called instance methods.
* Methods often change the state of an object by changing the value of a property.

|  |  |
| --- | --- |
| Example: an FM Radio Object | FMRadio Class in Java Code |
| 1. Instance variables  * boolean isOn * double station * int volume  1. Instance methods  * setIsOn * setStation * setVolume | **public** **class** FMRadio {  **private** **boolean** isOn;  **private** **double** station;  **private** **int** volume;    **public** FMRadio() {  isOn = **false**;  station = 89.7;  volume = 6;  }    **public** **void** setOn(**boolean** isOn) {  **this**.isOn = isOn;  }  **public** **void** setStation(**double** station) {  **this**.station = station;  }  **public** **void** setVolume(**int** volume) {  **this**.volume = volume;  }  } |

Classes of the type discussed here commonly don’t have a main method, so they are not executable. **They are for data**, the Model in the MVC paradigm.

Class Declarations

* The instance variables and instance methods that define an object are defined by a class.
* A class is like a template that defines the properties and methods of objects of that class type.
* The class declaration only defines a template or blueprint for a class.
* An object of a class is called an instance of the class.

Analogy: think of the class declaration as a cookie cutter for creating cookie objects, and the cookies it creates as instances of the cookie class.

* A class defines the properties and methods for an object of that class.
* There can be only one public class in a file, and it must have the same name as the source code filename. The general form of a class:

public class SomeName {

instance variable declarations

constructor method declaration(s)

instance method declarations

}

* Figure 9.2 shows a class named Circle.
* Figure 9.3 shows how Circle might be coded in Java.
* Note there is no main method in this class.

Introduction to UML (Unified Modeling Language)

* Unified Modeling Language™ (UML) is an industry-standard language for specifying, visualizing, constructing, and documenting the parts of software systems. It helps in the design of Java classes. Liang has many in our book.
* This object-oriented system of notation has evolved from the work of Grady Booch, James Rumbaugh, Ivar Jacobson, and the Rational Software Corporation.
* Today, UML is managed by the Object Management Group (OMG) as the standard for modeling object oriented programs. It simplifies the complex process of OO software design, making a "blueprint" for construction.
* UML defines nine types of diagrams: class (package), object, use case, sequence, collaboration, statechart, activity, component, and deployment.
* UML class diagrams offer a concise way to describe the composition of a Java class.

**Employee**

-name : String

-salary : double

+Employee( nm : String )

+toString( ) : String

+getSalary( ) : double

+setSalary( sal : double )

Class name

public class Employee {

private String name;

attributes

(properties)

private double salary;

public Employee ( String nm ) { … }

operations

(methods)

public String toString( ) { … }

public double getSalary( ) { … }

public void setSalary( double sal ) { … }

}

* See Figure 9.4 on page 324 for the UML diagram for the Circle class described earlier.

Examine class SimpleCircle on page 326-327, then try TestSimpleCircle

* Java programmers don’t commonly put a main method in data class, although it is possible as SimpleCircle demonstrates. Instead, programmers typically code data classes and main classes in separate files.
* The next TV example illustrates this practice.

Examine class TV on page 327-328. Note the UML, too. Then try TestTV

Constructing Objects Using Constructors (See page 329)

* Constructors are special methods used to create objects of a class.
* One or more special methods called constructors (abbreviation ctors) are also normally included in the definition of a class to assign initial values to the instance variables.
* **Ctors** are **public** and have no return-type. Ctors must have the same name as the class.
* Ctors with parameters set the properties of an object when it is instantiated with new in a program.
* A no-argument ctor can be added so an instance can be created with default values.
* A default ctor is added automatically by Java if no ctor is defined.

Accessing Objects via Reference Variables (page 330)

* The name of a class is used to create a reference variable for an object of that class type.

Circle circ;

* The new operator can then create a new object of the class and assign it to the reference variable.

circ = new Circle(); // in this case, this is a no-args ctor

* The two steps above can be combined.

Circle circ = new Circle( ) ;

Accessing an Object’s Data and Methods

* The dot operator is appended to the reference variable to accomplish these tasks.
* You have seen this many times already in this course.

Exercise

Write a data class named **Rectangle.java** with double properties for the **length** and **width**. Add a no arg ctor that sets both properties to zero and a parameterized ctor. Include two methods to return the perimeter and area of a rectangle, respectively. Add a main method to Rectangle.java and use it to instantiate several rectangles and display their areas and perimeters. In a separate file, create a program named **TestMyRectangle.java** that instantiates several rectangles and displays their areas and perimeters.

#### Difference Between Variables of Primitive Types and Reference Types (page 332)

* The variable of primitive types such as ints, doubles and chars stores the actual value of the variable. Object types are stored differently.
* When an object is instantiated, the object variable doesn’t hold the object, it holds a reference to the object. A reference stores a value indicating the address of the instance.
* Any object variable can be assigned the value null,a Java reserved word.
* A null is used to initialize an object variable without creating an object.
* An instance method cannot be accessed from an object with a null value.
* More than one object variable can refer to the same object.

Introduction to the Date Class (page 334-335)

* The Date class is available in the **java.util** package.
* View the skeletal UML diagram for the Date class on page 334.

The Random Class

* View the UML diagram for this class on page 334.
* Note the point about **seeding** the ctor.
* The default no-args ctor uses the time as a seed, should generate differently every time.

### The Point2D Class

* This class in the Java API was developed to represent a point in a two-dimensional graph.
* The UML diagram on page 336 shows some of its methods.

Try TestPoint2D

### Static (aka class) Methods and Properties, Constants

Static Methods

* A class, or static, method is one that can be called without creating an instance of the class.
* A static method belongs to the class itself, not to any instance of the class.
* The Math class with methods like Math.sqrt( ) and Math.max() is an example.
* In fact, all of the methods of the Math class are static methods.
* The static modifier is required to create a static method.

**public static int getCalories ( ) {**

**return calories;**

**}**

* A static method cannot access instance variables of the class, only static variables (covered next) and its own parameters.
* A static method can take parameters and return a value.
* If a static method is declared public, the method can be called with the class name and the dot operator.
* The main( ) method (which is static) cannot invoke a non-static method.
* You cannot create an instance of the Math class, you can only execute its static methods such as:

**Math.sqrt() Math.round() Math.pow()**

* The main method in an executable class is a static method.
* A static method intended for the internal use of a class should be private.
* To make static methods accessible to other classes, declare them as public.

### Static (class) Variables

* Every new object of a class gets its own set of instance variables.
* Static variables belong to the class, but not to any one instance.
* Static variables are stored only once in a program, and are accessible to all instances.
* As with static methods, the static keyword is used to create static variables.
* A public static variable can be accessed by the **class name** and the dot operator.

System.out.print( Merchandise.numInStock );

* static variables are used for:
  + Variables accessible to all methods in all classes (if public).
  + Storing data for the use of static methods.
  + For constants. To do so, include the keyword final in the declaration:

public static final double SALES\_TAX\_RATE = 0.07;

By convention, constants are named in upper case with underscores as shown.

Examine CircleWithStaticMembers on page 338.  
Then try TestCircleWithStaticMembers

### Visibility (aka Access) Modifiers (page 342-344) Package Access

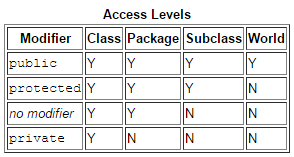
### When no visibility modifier is used for a property or method, it is accessible by any class in the same package.

The declaration of an instance variable, method, or ctor normally starts with an access modifier:

public - means can be accessed by all classes.

private - limits access to only the current class and its methods.

protected - limits access to the current class and subclasses of it (more later).



* Commonly, instance variables are declared as private and instance methods and constructors are declared as public. Declaring instance variables as private is a security measure termed information hiding, or data field encapsulation. This safeguards your object data from other classes.
* A method used to retrieve the value of an instance variable is called an accessor, or getter.
* A getter has the name pattern **getPropertyName()** and returns the desired property.
* A method that sets the value of a property is called a mutator, or setter.
* A setter always has the name pattern **setPropertyName(propertyType, propertyValue)**.
* A setter is always of return-type void and always needs a parameter for the assignment.

Examine CircleWithPrivateDataFields on page 345-346.  
Then try TestCircleWithPrivateDataFields

Example:

Needs parameter for the assignment

**public class SomeClass {**

**private int quantity;**

**public void setQuantity( int qty ) {quantity = qty;} //mutator**

**public int getQuantity() {return quantity;} //accessor**

Overloading Constructors

* Recall that a class may have multiple methods with the same name.
* This explains why a class can have two or more ctors.
* This overloading is legal as long as:
  + - The number of parameters is different.
    - The types of the parameters are different.

NOTE: The return type of a method is **not** sufficient for the compiler to distinguish between several methods with the same name.

### Objects as Method Parameters (page 347-351)

* Primitive arguments (ints, doubles, chars etc.) in methods are passed by value.
* This means that a method receives copies of the arguments for use within the method.
* If the method changes the value stored in the copy, this does NOT change the original value outside the method.
* This is not the case for methods that take objects as arguments. Object instances and arrays contain a reference (address of the object). They are NOT passed by value. They are passed by reference. The address is passed to the method.
* This means that when an object or array passed to a method is changed, this change will be evident after the method has finished executing.

Then try TestPassObject

Arrays of objects

* You can create arrays of objects in the same way as you create arrays of primitive types.
* All elements in the array must be of the same object type.

Then try TotalArea

### Immutable Objects

* These are objects that can be instantiated, but then not changed. The String class is immutable.
* To make your own immutable objects, make all properties private and don’t provide any setters.

The this Reference

* The this keyword, when used in a class definition, refers to the object itself.
* Note its usage on pages 356-357, especially when invoking a constructor.

### Choosing Method Names

* Start with a verb, since methods perform some action. For a method that computes a worker’s pay try:

public double **compute**( ) { … }

* If the verb doesn’t suffice, add a noun ( usually the target of the action )

public double **computePay**( ) { … }

* If the use of the method is still unclear, add adjectives for the noun

public double **computeWeeklyPay**( ) { … }

COP 2250 – Java Programming – Chapter 10 – Object-Oriented Thinking

As Liang states on page 366, this chapter is about the difference between procedural and object-oriented programming, with a focus on objects and class design.

Class Abstraction and Encapsulation

* Abstraction refers to the distinction between using a class and its implementation.
* Put another way, one can use a class without knowing the details of how it was coded.
* The implementation can be hidden from users, or encapsulated in a “black box”.
* Why would this encapsulation be desirable?

Examine class Loan on page 368-369. Note the UML diagram, too. Then try TestLoanClass

Thinking in Objects

* Read section 10.3 on page 370 to learn the benefits of thinking in objects.

Examine class BMI on page 371-372. Note the UML diagram, too. Then try UseBMIClass

Class Relationships

* Understanding the principles of relationship is useful when designing an OOP program.
* The key relationships are association, aggregation, composition, and inheritance (studied later in this course)

The Association Relationship

* This is a binary relationship because it describes activity between two classes.
* Association is defined as a structural relationship that conceptually means that the two components are linked to each other.
* This kind of relation is also referred to as a using relationship, where one class instance uses the other class instance or vice-versa, or both may be using each other.
* The lifetime of instances of the two classes are independent of each other and there is no ownership between two classes.
* See page 373 for a discussion and UML diagram for association.
* Refer to Figure 10.5 to see how association is implemented in class fields and methods.

Aggregation and Composition

* Aggregation is like association but with an additional point that there is an ownership.
* One object can “own” one or more other objects. This is called aggregation.
* The owner is the aggregating class. The class owned is the aggregated class or object.
* Aggregation is characterized by the “**has a**” relationship.
* If the aggregating class has exclusive ownership of the aggregated class, the relationship is said to be composition. A Car object “has a” Motor object, for example.
* Figure 10.6 shows a UML diagram used for these relationships.
* When we study inheritance in Chapter 11 we will also encounter the “**is a**” relationship.

Examine class Course on page 377. Note the UML diagram, too. Then try TestCourse

Designing a Class for Stacks

* See the Case Study on page 378.

Examine class StackOfIntegers on page 379 and the UML diagram on page 378, too. Then try TestStackOfIntegers

Processing Primitive Data Type Values as Objects

* Primitives like int, double, char, etc. are not object types, but each has its corresponding **wrapper** type in the Java API.
* Note the UML diagrams for the Integer and Double wrapper classes on page 381.
* Note the methods on page 381-382
* The static **valueOf()** method in all wrappers creates a wrapper instance from a String.
* The “parse” methods can parse numbers into **different number systems** with a specified **radix**.

Automatic Conversion Between Primitive and Wrapper Types

* This is known as boxing and unboxing (page 383-384).
* In **autoboxing**, a wrapper object is created with simplified syntax (top page of 384).
* Wrapper data can be used with the same syntax as with primitives in **autounboxing**.

BigInteger and BigDecimal

* The **java.math** package has these classes for extra-large and very precise numbers.
* There is no upper limit for the size of a BigInteger instance and no limit to the precision of a BigDecimal instance.

Try LargeFactorial

The String Class

* Note the three ways to create a String object on page 386.

String Class Methods

* Study the UML diagrams and examples for these String method groupings:
  + Replacing and Splitting Strings: see page 387-388.
  + Matching, Replacing, and Splitting with Patterns: see page 388-389.
  + Conversion between Strings and Arrays: see page 389-390.
  + Formatting Strings: see page 390.

StringBuilder and StringBuffer Classes

* These classes were developed to get around the issue that a String is immutable.
* Objects of these classes can be altered by editing and appending.
* The two classes are essentially the same except that StringBuffer methods are synchronized and therefore thread-safe.
* Note the UML diagram and examples for StringBuilder on pages 393-394.
* Note also the additional StringBuilder methods in the UML diagram on page 395.

Palindromes Case Study (see page 396)

* Note the steps needed to check a palindrome by ignoring alphanumeric characters.

Try PalindromeIgnoreNonAlphanumeric

**COP 2250 – Chapter 12 – Exception Handling and Text I/O**

Exception Handling

* Unless coded to prevent it, runtime errors in a Java program will cause the program to crash and present a stack trace to the user. Java will throw an **exception**.

Try **Quotient,java**

* As a coder, you could deal with this problem with a simple if…else block as shown next.

Try **QuotientWithIf**

* You could also write a method to cope with the possible runtime issue.

Try **QuotientWithMethod**

A better method would throw an exception that could then be handled by the calling method.

Try **QuotientWithException**

* Study closely the explanation of this latter example on page 453. Here is a summary:
* The code that might throw the exception (the call to **quotient**) is put in a **try block**.
* Remember the **try** block code might execute without throwing an exception, too.
* A selection inside quotient is coded to throw an ArithmeticException if division is by 0.
* The **catch** block executes if the exception is thrown, handling it gracefully.
* The catch block takes a **parameter** of **type** ArithmeticException.
* This catch block parameter object has methods you can use (especially **getMessage()**).
* The **getMessage()** method will return the string supplied with the throw statement.
* As noted atop page 454, a method separates detection of an exception from handling it.
* Many methods in the Java API have been coded to throw exceptions.
* Note the nifty use of the **do…while** loop and the **Boolean** variable in the next example.

Try **InputMismatchExceptionDemo**

**Exception Types**

* Exceptions are objects, and they all inherit from the superclass **Throwable**.
* Examine Figure 12.1 showing the inheritance hierarchy of **Throwable** and **Exception**.
* You can extend class Exception, or one of its subclasses, to make your own exceptions, too.

Three types of exception classes:

1. System Errors

* Class **Error** describes these.

1. Subclasses of Exception

* IOException, FileNotFoundException, and dozens more including RunTimeException (covered next). See class Exception in the Java API.

1. Subclasses of RuntimeException

* ArithmeticException, NullPointerException, IndexOutOfBoundsException, etc.

Examine Tables 12.1, 12.2, and 12.3 on pages 456-457 to get familiar with some common exceptions and the situations that will generate them.

Unchecked and Checked Exceptions

* RunTimeException, Error, and their subclasses are **unchecked** (by the compiler).
* If you miss these when coding, they will become apparent at runtime.
* All other exceptions are **checked** exceptions.
* The compiler, and a good IDE, forces you to handle these checked exceptions.
* The preferred technique for handling checked exceptions is by writing try-catch blocks.

More on Exception Handling

* Java handles exceptions with three operations:

1. Declaring an exception

* Every method must declare, in a comma delimited list, all of the exception types it might throw with the **throws** keyword. This is added to the method’s header.

1. Throwing an exception

* The **throw** keyword is used to throw an exception of the appropriate type if an error is detected.
* The new keyword is used to run the exception’s constructor, often with a string argument.
* If an instance of the exception class is specified (see Note on p459), the **getMessage()** method can be used to retrieve the ctor string.

1. Catching an exception

* This done with a **try** block and as many **catch** blocks as needed to handle the possible exception types that might be thrown.
* If the try block throws an exception, each catch block is examined in an attempt to find a match for the exception type.
* If none is found, the exception is passed to the main method and an error message is displayed in red on the console (and we know about that don’t we?☺).
* If multiple catch blocks are required, they should be ordered with subclasses of Exception first.
* Study the scenario on page 460 and examine Figure 12.3. Note the bullet list, too.

Getting Information About an Exception

* The object thrown by an exception holds data about the exception.
* Figure 12.4 on page 461 shows a UML diagram with some useful methods of the exception object that you can use.
* Especially useful are **getMessage(), printStackTrace(),** and **toString()**.

Try **TestException**

Examine **CircleWithException** and try **TestCircleWithException**

The finally clause

* A **finally** block can be optionally included after the catch block(s) if you want some code to execute if an exception is thrown, or not.
* The code in a finally block will always be executed.
* Good usages for the finally block include closing open files and open network connections.

When to Use Exceptions

* Simple exceptions should be handled with try-catch blocks right in the method.
* Don’t use exceptions where simple selections logic is sufficient.

Rethrowing Exceptions (see page 468-4690

* Rethrowing an exception gives other handlers in the caller a shot at handling the exception.

Chained Exceptions

* A second exception can be thrown along with the original to provide additional information.

Custom Exception Classes

* You can extend the Exception class to make custom exceptions for your application.
* You might need an exception to adhere to a business rule, for example.

Examine **InvalidRadiusException** and try **TestCircleWithCustomException**

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> Text I/O <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

The File Class

* This class was created in the API so an object can represent a file.
* The class contains methods for file properties and file manipulation.
* The UML diagram on page 474 shows the names and usages of common File class methods.
* Text files are ASCII files that can be read with programs like Notepad or WordPad.
* The File class creates File objects and has methods for file properties, but reading and writing with files requires other Java classes.

Try **TestFileClass**

File Input and Output

* The **PrintWriter** class in the **java.io** package can be used to write data to a plain text file.
* View its methods on page 476.

**T**ry **WriteData**

Note this line (line 2) in WriteData.java:

public static void main(String[] args) throws Exception {

* Java programs that work with files can encounter issues. When they do, they throw an **Exception**, so we must add the phrase “throws exception” here.

Try **WriteDataWithAutoClose**

Reading A File with Scanner

* The **Scanner** class used earlier in this course can read disk files as well as keyboard input.
* Java, like some other languages, considers files and the keyboard as similar entities.
* Both are viewed as possible sources of a stream of characters.
* Review the methods of Scanner in the UML diagram on page 478.

**T**ry **ReadData**

* We need “throws Exception” here too (see line 4), since the file may not exist etc.
* Scanner class and some skills for it are explained very well on pages 479-482.
* Scanner can be used to replace text in a file. See page 480.

**T**ry **ReplaceText**

Reading Data from the Web

* A Java program can read a file from across the Internet, if the URL of the file is known.
* The java.net.URL class is used to create an object for connecting with the remote resource.
* The openStream() method of the URL object is specified in the Scanner’s ctor to make the connection. Then a loop is used to read the remote file line-by-line.
* The connection could fail for a number of reasons, so several exceptions must be handled.

**T**ry **ReadFileFromURL**

**T**ry **WebCrawler**