Introduction to Java Programming

Student Handbook

Themis
Leaders in IT Education

Introduction to Java

revision July, 2017

Overall Objectives

This course provides a detailed and thorough introduction to Java, including its compiled statements, the use of classes and objects, and the use of some classes of the Java API.

Upon successful completion of this course you will be able to read, modify and create Java programs using compiled statements and some classes in the API. You will be able to use Java documentation to learn more packages and classes.

Table of Contents

Introduction to the Java Language

Statements

Methods

Classes and Objects

Appendix: Optional Topics

revision July, 2017

UNIT 1: OVERVIEW OF JAVA

Upon completion of this unit, students should be able to:

1. Briefly describe the steps to compile and execute a Java application.

2. Locate web pages to download the Java Development Kit, Java tutorials, and javadoc documentation.

TWO HISTORIES, TWO CONCEPTS OF INFORMATION

1884, Herman Hollerith
Worked for Census Bureau
Founder, company that became IBM

FIRSTNAME 1 – 20 JOHN	MIDNAME 21 – 40	LASTNAME 41 – 80 DOE

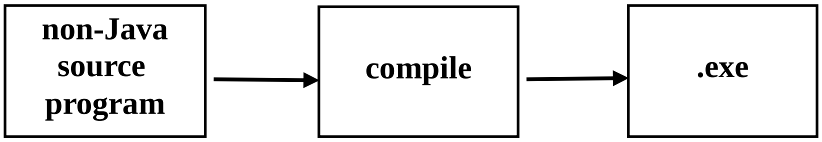
1876, Alexander Graham Bell
Inventor in Boston MA
Founder, Bell Telephone Co.

```
\njohn::doe\n
```

1.	The meaning of <u>fields</u> is determined by the space they occupy and the location of the space.	The meaning of <u>words</u> is determined by their sequence within a line.
2.	A field is one or more consecutive bytes reserved for a specific item of data.	A word is zero or more consecutive bytes between separator characters.
3.	When you have no significant data, the field retains its space and location, and contains fill characters.	When you have no significant data, the word is zero length.
4.	SPACE LOCATION	TIME SOURCE and DESTINATION
5.	Data resides	Data travels.
6.	File Record Field	Stream Line Word

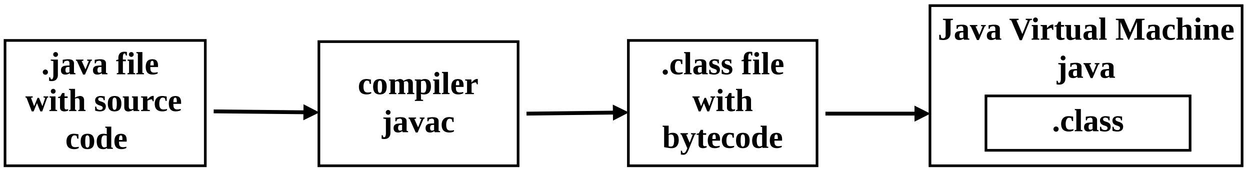
NOTE:
\n is the newline character, used as line separator.
: colon is a commonly-used word separator within lines.

TWO-PART COMPILE PROCESS



1. The compiled executable works only in the one platform it is compiled for.

2. The executable is an independent file that can execute without its compiler.


.java file with source code
compiler javac
.class file with bytecode
Java Virtual Machine java
.class

3. Java source code files must have the .java filename extension. The compiler is called javac. Compiling produces bytecode in a file with the .class filename extension.

4. Bytecode is partially-compiled code (the compiler does not resolve platform-dependent parts of the source code). Below, only partial output of the dir command is shown.

```
    C:\myjava> dir
    02/21/2017  11:40 AM          129 MySourceCodeFile.java

    C:\myjava> javac MySourceCodeFile.java

    C:\myjava> dir
    02/21/2017  11:46 AM          431 MySourceCodeFile.class
    02/21/2017  11:40 AM          129 MySourceCodeFile.java
```

5. Bytecode is executed within the Java Virtual Machine (JVM). The JVM is an application program called java that pretends to be a computer. To execute your bytecode, execute java and give it the name(s) of your bytecode file(s) without specifying any filename extension.

```
    C:\myjava> java MySourceCodeFile
    hello
```

6. The bytecode program cannot execute independently outside of the JVM. The JVM enforces security and provides an architecture-neutral execution environment.

7. All java code is platform-independent except the Java Virtual Machine (JVM). The JVM is ported to its platform.

OPTIONAL: HISTORY

1. Java arises from the programming tradition that started with C Language and C++.

    a. C Language is a structured, high-level language that was created to replace assembly language for coding the UNIX operating system. C is small and efficient, with a large library of pre-written standard functions. C also has standard "header files" that allow it to be portable from one platform to another, but portability is something the programmer has to work for. C is designed for I/O with files, and with standard input, standard output, and standard error, which are streams relating to a console keyboard and monitor screen.

    b. C++ is an object-oriented superset of C Language. To achieve portability, C++ still needs a compiler for the target CPU on which the program will run.

2. Java was created at Sun Microsystems, Inc., starting in 1991. Originally called Oak, it was renamed Java in 1995. The designers needed an architecture-neutral programming language for programming consumer electronic devices such as portable telephones and microwave ovens. The emergence of the World Wide Web at the same time precipitated Java into prominence, because with the WWW, programs cannot be compiled in advance for their target CPU, because until the program is downloaded no one knows where it will execute.

    a. Java applets are programs that execute within a Java-enabled web browser, and allow a web page to be interactive without requiring server involvement.

    b. Applets are no longer as important as they were in 1995 because:
        i. Not all browsers implement Java properly.
        ii. Other tools have become available for achieving similar results.

3. Like C and C++, Java was created by programmers for themselves. Like C and C++, Java provides the control, flexibility, and power that allow programmers to create the best solutions for the programming problems they work on.

4. In 2010 Oracle Corporation acquired and merged with Sun Microsystems, Inc. to become Oracle America, Inc. Oracle now provides the Java language, tutorials, and documentation for free over the internet.

OPTIONAL: BYTECODE IS ARCHITECTURE NEUTRAL

PROBLEM: Platforms vary in their hardware and software.

1. If you compile a program into the native binary machine code, the executable is tied to that specific hardware.

2. Each operating system has its own unique API ("Application Programming Interface", that is, function library) especially for graphics. For example, different operating systems' APIs might require different parameters in order to create a window, accept user input, or draw on the screen.

SOLUTION: Java compiles the program into architecture-neutral bytecode, which can be executed in any computer that has a Java interpreter, called the Java Virtual Machine or JVM. The JVM interprets the bytecode for its specific CPU and native API.

3. The JVM interprets bytecode to execute in the local platform, so that the same bytecode from a server can be downloaded into different clients and executed.

4. The JVM is specific to its platform. Like an interpreter, it goes line by line. Like a compiler it performs translation (but it does not save the executable).

5. Java provides its own API, so bytecode is independent of the API of the local operating system. The JVM converts the Java API into the native API.

SPEED OF EXECUTION

6. Because of the prior compile into bytecode, Java programs execute faster than if they had to be completely compiled just prior to execution. Execution of bytecode is slower than execution of a program that is already in the native binary machine language, but a binary program can't be platform independent.

    a. JVMs may apply "Just In Time" dynamic compilation to the bytecode, achieving execution performance similar to compiled code.

    b. Some development and execution environments may save a Java executable, but this would be very unusual and rare.

OPTIONAL: BYTECODE IS SECURE

1. The JVM retains control of the executing program, providing it with a runtime environment and confining it to that environment, preventing it from creating unwanted side effects outside of the runtime environment. Java does not create a binary executable file that can run independently. The JVM is always needed in order to execute a Java program.

2. The bytecode cannot be tampered with, because the JVM checks the bytecode internally to detect modification since its compilation.

3. Java pointers are not accessible to the programmer. Java uses "references" which are pointers that a programmer is not allowed to directly dereference or manipulate.

4. The security of the runtime environment allows Java applets to be secure. Applets are small Java programs that can be dynamically downloaded across a network, and execute in a Java-enabled Web browser. Applets cannot execute as independent programs. The JVM limits them in these ways:

   a. applets cannot initiate execution of another program.
   b. applets cannot open files.
   c. applets cannot open a connection to a URL other than the one they were downloaded from.

DOWNLOAD JDK, VIEW TUTORIALS OR JAVADOC

1. Two types of Java downloads are:

    a. The JRE (Java SE Runtime Environment) allows end-users to run Java applications, but does not contain the compiler or other development tools.

    b. The JDK (Java SE Development Kit) includes the JRE and also has the commandline development tools, such as the compiler, that are needed or useful for developing applications.

2. The JDK can be downloaded free from Oracle. To find the web page do a web search on "Oracle Java 1.8 JDK download"

3. To find an Oracle Java tutorial on a specific topic, do a web search on "Oracle java tutorial yourtopic".

4. The Javadoc 1.8 is documentation for the Java API (Application Programming Interface), which is the standard library of pre-written classes that provide code for common programming tasks such as working with Strings, performing math functions, networking, database connectivity, etc. To find the javadoc, do a web search on "javadoc 1.8 API".

OPTIONAL: JAVA FEATURES AND COMPONENTS

1. Java code can be used in:
    a. Stand-alone application programs
    b. Applets that run within a web page in a browser
    c. Servlets within a servlet container on a server
    d. JSP Java Server Pages embedded in HTML
    e. EJB Enterprise JavaBeans

2. Features:
    a. Source code and bytecode are platform independent
    b. Object oriented (all code must be in a class)
    c. Expression syntax inherited from C and C++
    d. Automatic garbage collection
    e. Use of references rather than pointers
    f. Built-in support for multi-threading
    g. Exception handling via throw, try, and catch

3. Java has a large API (Application Programming Interface) which is a standard library of pre-written classes with support for networking, database connectivity, etc. The API specification for Java 7, Standard Edition, lists 4025 pre-written classes. Java 8 has 4240.

4. Components include:
    a. Compiler-defined elements
        1) Basic (built-in) data types, variables, and literals
        2) Operators
        3) Flow of control conditional and looping structures
        4) Reference variables and objects for arrays, Strings, and all other class types
    b. User-defined types, known as classes, which you create
    c. Pre-defined types, which are the classes in the API, which are organized in packages such as java.lang, java.io, java.math, java.net, java.text, java.util, etc. Packages are usually implemented as folders.
    d. Javadoc documentation

UNIT 2: Hello.java APPLICATION

Upon completion of this unit, students should be able to:

1. Create and execute a simple Java application program that includes comments.

2. Briefly describe the compile process for Java applications.

NOTES FOR Hello.java

1. A source file must have a name that consists of the name of the public class followed by the .java filename extension.

2. A source file for an application usually contains one class. It is unusual for a file to contain more than one class, but if that happens, only one class can be public, and the file must be named for that public class.

3. Java is case sensitive! Even if your operating environment treats Hello1.java and HELLO1.java as the same, Java won't.

4. A named, callable piece of code is called a "method" rather than a subroutine or function.

5. An application must have one method called main which is where execution begins.

6. Methods must be inside a class. A class is the compilable unit of code.

7. The method main always has three modifiers:
    a. public — A public method can be called by any other method in your program. The main method is public so it can be called by the JVM.
    b. static — A static method can be called even if an object of its class has not been instantiated.
    c. void — A void method does not return a value.

8. System.out.print and System.out.println are methods that display text on the screen.
    a. print — print argument(s) as is
    b. println — print argument(s) and add a newline at the end

9. Java is free-form, but code conventions should be followed for readability. Words may be separated by spaces, tabs, or newlines.

10. Each statement must end in ; semicolon.

11. // starts a comment which goes to the end of the line.

Hello.java APPLICATION PROGRAM

Hello1.java

```
public class Hello1 {
    public static void main (String[] args) {
        System.out.println ("Hello java");
    }
}
```

Result, Hello1.java

```
Hello java
```

Hello2.java

```
//Version 2

public class Hello2                         //class header
{
    public static void main (String[] args) //method header
    {
        System.out.println ("Hello java, ");
        System.out.println ("version 2");
        System.out.println ("Separate print line");
    }
}
```

Result, Hello2.java

```
Hello java, version 2
Separate print line
```

Hello3.java

```
public class Hello3{public static void main(String[]args)
{System.out.print("Hello java, ");System.out.println
("version 3");System.out.println("Separate print line");}}
```

Result, Hello3.java

```
Hello java, version 3
Separate print line
```

OPTIONAL: THREE KINDS OF COMMENTS

P204.java

```
1   /**
2   * Documentation comment for the class
3   * must be immediately above the class header.
4   * The asterisks on line 2 through 6 are not required,
5   * and will not show up in your javadoc.
6   * The class comment is often a tutorial about the class.
7   */
8
9   public class P204 {
10
11      /**
12      * Documentation comment for a method
13      * must be immediately above the method.
14      */
15
16      public static void main (String[] args) {
17
18          //Single-line comment goes from // to end of line
19          System.out.println ("use single-line comments");
20
21          /*
22             Multi-line comments CANNOT BE NESTED
23             and are infrequently used
24          */
25          System.out.println ("enter comments as you code");
26      }
27  }
```

Result, P204.java

```
use single-line comments
enter comments as you code
```

1. Three forms of comment:
    a. /** documentation comment, cannot nest */
    b. // single-line comment to end of line
    c. /* multi-line comment, cannot nest */

2. Programs are more readable if you use // all the time. When you need to comment out a section of code, use the multi-line comment /* */

3. Documentation comments are used by the javadoc documentation generator to create documentation with the same form and appearance as Java's API documentation.

EXERCISES

1. Follow the Eclipse instructions on page 2.07 to create the program E21.java by entering the solution program on page 2.06. Compile and execute the program.

2. Create a program called E22.java that prints your name and job title on separate lines. Compile and execute your program.

3. Revise your program E22.java and call the modified version E23.java. Include at least two blank lines in your program. Make your name and job title display in the center of the screen. (Hint, use \n for the newline and \t for the tab.)

4. To develop familiarity with error messages and debugging, revise your program E22.java and call the revised version E24.java. ONE AT A TIME, make the following errors in the program and try to compile and execute it. Correct the error before you go ahead to experiment with the next error.

   a. Change the name of the class to have a lower case e, but keep the filename of the program with an upper case E.

   b. Omit the ; semicolon at the end of the first print or println statement.

   c. Spell the keyword class with an upper case C, as in Class.

   d. Spell the name main with an upper case M, as in Main.

   e. Spell println incorrectly, as prrrtln, in one statement.

   f. Omit the prefix System.out. with the name println in one statement.

   g. Omit the ( ) parentheses with one println statement.

   h. Omit the closing } curly brace in the file.

   i. Omit the keyword static in the header of main.

   j. Omit the keyword public in the header of main.

SOLUTIONS

E21.java

```java
public class E21 {
    public static void main (String[] args) {
        System.out.println ("E21 says Hello!");
    }
}
```

E21.java

```
E21 says Hello!
```

E22.java

```java
//Exercise E22.java

public class E22 {

    public static void main (String[] args) {
        System.out.println ("teresa\nteacher");
    }

}
```

Result, E22.java

```
teresa
teacher
```

E23.java

```java
//Exercise E23.java
public class E23 {
    public static void main (String[] args) {

        System.out.println ("\n\n\n\n\n\n");
        System.out.println ("\t\t\tTeresa\n\t\t\tTeacher");
        System.out.println ("\n\n\n");

    }
}
```

Result, E23.java

```







            Teresa
            Teacher



                ---output ends here on the screen
```

ECLIPSE (based on Mars version)

1. If you are taking this course via BlackBoard Collaborate, before using Eclipse you must release the control-space keystroke combination in BlackBoard:

   a. In BlackBoard, click Edit, Preferences, HotKeys.
   b. Highlight "Take back control of application sharing".
   c. Click Modify. Change the keystroke combination to Control+Shift+Space.
   d. Click OK, Close.

2. If you downloaded the Eclipse zip file but have not extracted all the files yet, you must do that now.

   a. Using the zip download, Extract All into the folder C:\Eclipse.
   b. Use Windows Explorer to go to C:\Eclipse\eclipse. The icon for eclipse.exe is a blue sphere with three white lines across the middle. Make a shortcut on your desktop for eclipse.exe. IF YOU DRAG AND DROP THE ICON FOR eclipse.exe IT WON’T WORK.

3. <u>Launch Eclipse</u>: Click eclipse.exe or your shortcut for it.

4. <u>Make your workspace</u>

   a. The "Workspace Launcher" popup asks you to "Choose a workspace folder to use for this session". You may use the default. Write the full name of your workspace here:

      ____

   b. If the "Workspace Launcher" does NOT come up, after you enter Eclipse you can click File, Switch Workspace, Other. You can use C:\myjava\eclipse and do not click "Use this as the default and do not ask again". Write the full name of your workspace on the line above.

5. <u>Welcome Screen, Enter Eclipse</u>

   a. If you have NOT opened Eclipse before, you will get the Welcome screen. In the menu bar click Window, Perspective, Open Perspective, Java.

6. The Java Perspective arranges the screen and provides default code to assist in creating Java applications.

   a. The Eclipse title bar should say "Java - Eclipse".
   b. If the title bar says anything else, click Window, Perspective, Open Perspective, Java.

7. The Java Perspective is divided into 3 main sections

   a. The Package Explorer is on the left. It shows the folders and files that Eclipse creates for you. Java uses the term "package" for a folder or directory.

   b. The Editor is in the center. This is where you type in your source code.

   c. The Outline view and Task List are on the right. Close the Task List by clicking on the X in its tab.

   d. On the bottom of the Java Perspective you can close the sections called Problems, Javadocs, and Declarations by clicking on the X in their tabs.

8. Create MyProject
   a. Click File, New, Java Project.
   b. In the "New Java Project" popup, fill in MyProject for your project name (Eclipse will make the folder for it).
   c. Keep the selection of the square checkbox for "Use default location"
   d. For "JRE" keep the selection of the top round radio button "Use an execution environment JRE:".
   e. For "Project Layout" keep the selection of "Create separate folders for source and class files".
   f. Click Finish.
   g. The Package Explorer should now display your new project folder. Your src and bin folders are under it. Your bin folder will not be displayed in Package Explorer.

9. Create your class (CAUTION: Your screen must be tall enough to display the entire "New Java Class" popup window.)

   a. Highlight your project name in the Package Explorer. Click File, New, Class.

   b. For "Source folder" the name of your project followed by /src should already be filled in: MyProject/src

c. For "Package:" leave it blank.

d. For "Name:" enter the name of your new class: E21

e. For "Modifiers:" keep the selection of public

f. For "Superclass:" keep the selection of java.lang.Object

g. For "Interfaces:" leave the box blank.

h. For "Which method stubs would you like to create?" keep the selection of Inherited abstract methods AND click the checkbox for a main method. The label on this checkbox says "public static void main (String[] args)"

i. For "Do you want to add comments? leave it blank.

j. Click: Finish

10. Change the font in the Editor

a. Click Window, Preferences. In the Preferences popup, in the left panel, expand General. Then expand Appearance.

b. Click on Colors and Fonts. In the "Colors and Fonts" panel expand Java. Select "Java Editor Text Font (overrides default: Text Font)". Click the button Edit... on the right side to display the choices.

c. Consolas and Courier New are good. Select your choice of font, size, and bold versus regular. Click OK, OK.

11. In the Editor, delete the comment template in main.

12. In the Editor, inside the main method, type in the statement:

```
System.out.println ("E21 says Hello!");
```

13. Line Numbers

a. Right-click in the light-blue left margin bar. In the popup, select the option to show line numbers.

b. While the Editor "has focus" the information bar at the bottom of Eclipse shows the cursor position line:column.

14. Execute your program while your class is displayed in the Editor and the Editor has focus.

    a. The run icon is a green circle with a white triangle.

    b. If you have not saved, when you click the run icon, the "Save and Launch" popup asks which resources to save, and offers a checkbox for "Always save resources before launching".

        1) If you click that checkbox, Eclipse will save automatically before compiling and executing.

        2) The Editor allows you to undo via CTRL-Z after you save, compile, and find errors.

    c. Console output from System.out.println is displayed in the Console view below the Editor.

15. Content assist

    a. Inside the main method, create a new System.out.println() statement by typing sysout and pressing control-space.

16. Console view

    a. The Console view becomes visible below the Editor when you run an application that creates output from System.out or System.err.
    b. You can raise or lower the Console view height via your mouse by dragging and dropping the Console upper border.

    c. Maximize: When the Console tab appears below the Editor, double-click the tab.
    d. Minimize: Double-click the Console tab.
    e. Close: Click on the X button.
    f. Open: Click Window, Show View, Console.

17. Problems view

    a. The Problems view shows errors and warnings in the Editor after you save.

    b. After you fix an error, its red circle with X gets white or gray; after you save again the circle goes away.

    c. Display Problems view: Click Window, Show View, Problems.

UNIT 3: BASIC DATA TYPES: LITERALS, VARIABLES, OPERATORS

Upon completion of this unit, students should be able to:

1. State the rules and conventions for identifiers.

2. Briefly describe two reasons why data type is important.

3. Declare variables of the basic types, both with and without initial values.

4. Assign the value of a literal or an expression to a variable of a basic type.

5. Briefly describe the purpose of the final modifier, and use it to create symbolic constants.

6. Optional. Briefly describe the purpose of the cast operator and when it is needed, and use it in assignment expressions.

IDENTIFIERS AND KEYWORDS

1. An identifier is the name for a variable, method, class, or interface. Rules for identifiers are:

   a. First character must be   a-z A-Z _ $
   b. Other characters must be   a-z A-Z _ $ 0-9
   c. Cannot be the same as a keyword
   d. No length limit
   e. Advice: Do not start your identifiers with $ to avoid collisions with compiler-generated names.

2. Code conventions:

   a. Variable and method names:

```
myVariableIdentifier
myMethodIdentifier()
```

   b. Class and interface names:

```
MyClassIdentifier
MyInterfaceIdentifier
```

   c. Final variables:

```
MY_SYMBOLIC_CONSTANT
```

3. An identifier cannot be the same as a keyword. Keywords are:

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

4. Keywords const and goto are not in current use but are reserved for future use. Some authorities do not consider true, false, and null to be keywords.

DATA TYPES

1. The data type of a variable determines:
   a. The values that the variable can contain.
   b. The operations that can be performed on the variable.

2. A variable has either basic or class type. Class types will be covered later in this course.

3. A variable of basic data type contains its value, which is a single value. The basic data types are:

a.	four sizes of integer	byte, short, int, long
b.	two sizes of floating-point	float, double
c.	two-byte character	char
d.	boolean, holds only true or false	boolean

4. Two types of statements are declarations and procedural statements. Declarations start with the name of a data type.

```
int roomCount = 1;               //declaration statement
roomCount = roomCount + 1;       //procedural statement

TCourse6 tc;                     //declaration statement
tc = new TCourse6 ("Java", 12); //procedural statement
```

OPTIONAL

5. Java is a strongly-typed language.
   a. Each basic data type is defined unambiguously.
   b. Each variable and expression has a type.
   c. javac checks assignments for type compatibility.
   The cast operator can request "reasonable" conversions.
   Casting is an optional topic in this unit.

6. Some authorities consider char to be an integer, but others consider it to be a separate category, because although it can be used for its integer numeric value its significance is the character it signifies. Also, all other numeric types are signed, but char is unsigned.

7. Arithmetic with basic types can be done on integer and floating-point data types only (the value in char variables can be manipulated via arithmetic, but requires casting).

8. Arithmetic types have internal representations that are platform independent, so arithmetic results will be identical regardless of what platform the program runs on.

INTEGER DATA TYPES

P304.java

```
public class P304 {
    public static void main (String[] args) {

        byte  b = 1;           //declare byte var called b
        short s=2;             //declare short var called s
        int   i=0, j, k=-4;    //declare three int vars
                               //local var j contains garbage
        long  tot = 12345L;    //declare long var called tot

        tot = b + 24 + s + k;  //procedural statement
                               //When + is between 2 numbers
                               //it means add

        System.out.println ("tot=" + tot); //The + between a
    }                          //String and any other type
}                              //means concatenate to String
```

Result, P304.java

```
tot=23
```

==

1.

name	bits	bytes	signed?	range
byte	8	1	yes	-128 to 127
short	16	2	yes	-32,768 to 32,767
int	32	4	yes	-2,147,483,648 to 2,147,483,647
long	64	8	yes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

2. Integer literals are stored as int by default. If the letter L or l is appended, the literal is stored as a long.

3. Initialization in a declaration can be done with constants or with values computed during execution. Initialization in a declaration is performed when the declaration is executed.

OPTIONAL

4. Negative numbers use two's complement notation. The leftmost bit is the sign; if it is 1, the number is negative.

5. Literals can be coded in decimal, octal, or hex. Negative values in octal or hex are coded in two's complement notation.

```
a.  decimal:      123    123L    -123
b.  octal:        0123   0123L
c.  hexadecimal:  0x123  0x123L       (0x or 0X, a-f or A-F)
```

INTEGER OPERATIONS

P305.java

```
public class P305 {
    public static void main (String[] args) {

        int i = 1 + 2 * 3 - 4 / 5;

        int q = 10 / 3;
        int r = 10 % 3;
        System.out.println ("i="+i + ", q="+q + ", r="+r);

        int j = 0;
        ++j;                       //j = j + 1;
        int k = 0;
        k++;                       //k = k + 1;
        System.out.println ("++j=" + j + ", k++=" + k);

        k = 4;
        i = 3 * ++k;               //k = k + 1;  i = 3 * k;
        System.out.print ("before: i=" + i + " k=" + k);
        k = 4;
        i = 3 * k++;               //i = 3 * k;  k = k + 1;
        System.out.println (", after: i=" + i + " k=" + k);
    }
}
```

Result, P305.java

```
i=7, q=3, r=1
++j=1, k++=1
before: i=15 k=5, after: i=12 k=5
```

1. Integer operations: (bit operations will not be covered)
    a. Comparison < <= >= > == !=
    b. Arithmetic * / % + -
    c. Increment and decrement ++ --

2. No indication is given when overflow or underflow occurs. Dividing by zero causes an ArithmeticException (exceptions are covered in an optional unit in the Appendix).

3. If ++ or -- is a PREFIX, ++ or -- is done BEFORE the other operation using the same variable in the same expression.

4. If ++ or -- is a SUFFIX, ++ or -- is done AFTER the other operation using the same variable in the same expression.

5. ++ and -- work only with variables. compileError = (i+j)++;

CHAR DATA TYPE AND UNICODE

P306.java

```
public class P306 {
    public static void main (String[] args) {

        char c1 = '3';              //character of digit 3
        char c2 = ' ';              //space character
        char c3 = '\'';             //escape sequence literal
                                    //for single quote
        System.out.println (c1 + "-" + c2 + '-' + c3);
    }
}
```

Result, P306.java

```
3- -'
```

1. A char is stored in two bytes (16 bits), is not signed, and uses the Unicode character set.

   a. The Unicode Worldwide Character Standard specifies characters for many languages. The first 128 Unicodes are the same as ASCII. The first 256 Unicodes are the same as the extended 8-bit ISO-Latin-1 character set. See www.unicode.org.

2. A char literal consists of the value of one character enclosed in single quotes.

3. The following escape sequences (also called backslash escapes) are available.

```
\' single quote     \f formfeed
\" double quote     \n newline
\\ backslash        \r carriage return
\b backspace        \t tab
```

OPTIONAL

4. A char literal may be specified three ways:
   a. Graphic symbol, 'A'
   b. Octal value '\ooo' in the range \000-\377, '\101' for 'A'
   c. Unicode '\uhhhh' in the range \u0000-\uffff , '\u0041' for 'A'

ASCII CHARACTER CODE

OCTAL

000 NUL	001 SOH	002 STX	003 ETX	004 EOT	005 ENQ	006 ACK	007 BEL
010 BS	011 HT	012 NL	013 VT	014 NP	015 CR	016 SO	017 SI
020 DLE	021 DC1	022 DC2	023 DC3	024 DC4	025 NAK	026 SYN	027 ETB
030 CAN	031 EM	032 SUB	033 ESC	034 FS	035 GS	036 RS	037 US
040 SP	041 !	042 "	043 #	044 $	045 %	046 &	047 '
050 (	051)	052 *	053 +	054 ,	055 -	056 .	057 /
060 0	061 1	062 2	063 3	064 4	065 5	066 6	067 7
070 8	071 9	072 :	073 ;	074 <	075 =	076 >	077 ?
100 @	101 A	102 B	103 C	104 D	105 E	106 F	107 G
110 H	111 I	112 J	113 K	114 L	115 M	116 N	117 O
120 P	121 Q	122 R	123 S	124 T	125 U	126 V	127 W
130 X	131 Y	132 Z	133 [	134 \	135]	136 ^	137 _
140 '	141 a	142 b	143 c	144 d	145 e	146 f	147 g
150 h	151 i	152 j	153 k	154 l	155 m	156 n	157 o
160 p	161 q	162 r	163 s	164 t	165 u	166 v	167 w
170 x	171 y	172 z	173 {	174 |	175 }	176 ~	177 DEL

HEXADECIMAL

00 NUL	01 SOH	02 STX	03 ETX	04 EOT	05 ENQ	06 ACK	07 BEL
08 BS	09 HT	0A NL	0B VT	0C NP	0D CR	0E SO	0F SI
10 DLE	11 DC1	12 DC2	13 DC3	14 DC4	15 NAK	16 SYN	17 ETB
18 CAN	19 EM	1A SUB	1B ESC	1C FS	1D GS	1E RS	1F US
20 SP	21 !	22 "	23 #	24 $	25 %	26 &	27 '
28 (	29)	2A *	2B +	2C ,	2D -	2E .	2F /
30 0	31 1	32 2	33 3	34 4	35 5	36 6	37 7
38 8	39 9	3A :	3B ;	3C <	3D =	3E >	3F ?
40 @	41 A	42 B	43 C	44 D	45 E	46 F	47 G
48 H	49 I	4A J	4B K	4C L	4D M	4E N	4F O
50 P	51 Q	52 R	53 S	54 T	55 U	56 V	57 W
58 X	59 Y	5A Z	5B [	5C \	5D]	5E ^	5F _
60 '	61 a	62 b	63 c	64 d	65 e	66 f	67 g
68 h	69 i	6A j	6B k	6C l	6D m	6E n	6F o
70 p	71 q	72 r	73 s	74 t	75 u	76 v	77 w
78 x	79 y	7A z	7B {	7C |	7D }	7E ~	7F DEL

DECIMAL

0 NUL	1 SOH	2 STX	3 ETX	4 EOT	5 ENQ	6 ACK	7 BEL
8 BS	9 HT	10 NL	11 VT	12 NP	13 CR	14 SO	15 SI
16 DLE	17 DC1	18 DC2	19 DC3	20 DC4	21 NAK	22 SYN	23 ETB
24 CAN	25 EM	26 SUB	27 ESC	28 FS	29 GS	30 RS	31 US
32 SP	33 !	34 "	35 #	36 $	37 %	38 &	39 '
40 (	41)	42 *	43 +	44 ,	45 -	46 .	47 /
48 0	49 1	50 2	51 3	52 4	53 5	54 6	55 7
56 8	57 9	58 :	59 ;	60 <	61 =	62 >	63 ?
64 @	65 A	66 B	67 C	68 D	69 E	70 F	71 G
72 H	73 I	74 J	75 K	76 L	77 M	78 N	79 O
80 P	81 Q	82 R	83 S	84 T	85 U	86 V	87 W
88 X	89 Y	90 Z	91 [	92 \	93]	94 ^	95 _
96 '	97 a	98 b	99 c	100 d	101 e	102 f	103 g
104 h	105 i	106 j	107 k	108 l	109 m	110 n	111 o
112 p	113 q	114 r	115 s	116 t	117 u	118 v	119 w
120 x	121 y	122 z	123 {	124 |	125 }	126 ~	127 DEL

FLOATING-POINT DATA TYPES

P308.java

```
public class P308 {
    public static void main (String[] args) {

        float  f = 1.5F;    //casting the literal is required
        double d = 2.7;

        d = d + f;
        System.out.println ("d=" + d);
    }
}
```

Result, P308.java

```
d=4.2
```

1.

name	bits	bytes	signed?	range
float	32	4	yes	1.40239846e-45 to 3.40282347e+38
double	64	8	yes	4.94065645841246544e-324 to 1.79769313486231570e+308

2. Floating point literals are double by default. If the letter F or f is appended, the literal is stored as float.
    a. float: 12.3f 45.67F
    b. double: 12.3 45.67 8.9E3

OPTIONAL

3. Internally, floating-point values are stored in IEEE 754 representation. This enables Java programs to produce the same result for floating point calculations in all platforms.

FLOATING-POINT OPERATIONS

P309.java

```java
public class P309 {
    public static void main (String[] args) {

        double result = 1.0 + 2.0 * 3.0 - 4.0 / 5.0;

        float  subtotal = 1.50F;  //cast operator is required
        double salesTax =  0.0825;
        double total = subtotal + (subtotal * salesTax);

        System.out.println ("r=" + result + ", t=" + total);

    }
}
```

Result, P309.java

```
r=6.2, t=1.62375
```

1. Operations that can be performed on floats and doubles are:

```
Comparison  <  <=  >=  >  ==  !=
Arithmetic  *  /  %  +  -
Increment and decrement ++  --
```

2. Operations with two floats are performed as float. Operations with a float and double are performed as double. Operations with an integer and a floating-point variable are performed in floating-point.

OPTIONAL

3. Floating-point operations never throw exceptions (exceptions are covered in an optional unit in the Appendix).

4. Casting is required to assign a double to a float, or any floating-point value to any other data type. No value can be cast to boolean. If you cast a float or double to an integer type, the fraction is truncated.

5. The remainder operator, if used with floating point numbers, provides the remainder after dividing only up to the decimal point:

```java
double remainder = 9.25 % 2;
System.out.println ("r=" + remainder);
```

result

```
r=1.25
```

BOOLEAN DATA TYPE

P310.java

```
public class P310 {
    public static void main (String[] args) {

        boolean mon = true;
        boolean tue = false;
        boolean wed = true;
        boolean thu = false;
        boolean fri = true;

        if (mon && tue && wed && thu && fri) {
            System.out.println ("1. five-day class");
        }

        if (mon && !tue && wed && !thu && fri) {
            System.out.println ("2. Mon, Wed, Fri class");
        }

        if (mon == true || tue == true) {
            System.out.println ("3. class meets Mon or Tue"
            + ", Monday is " + mon + ", Tuesday is " + tue);
        }
    }
}
```

Result, P310.java

```
2. Mon, Wed, Fri class
3. class meets Mon or Tue, Monday is true, Tuesday is false
```

1. The boolean type has only two values, represented by the keyword literals true and false.

2. The length of a boolean variable in the JVM during execution is not specified, and may be 1, 8 or 32 bits. When written to disk using the Serializable interface, a boolean is one byte.

3. Boolean operations:
    a. `==  !=`   Equal to, Not equal to
    b. `&&  ||`   Short-circuiting logical operations AND, OR
    c. `!`   NOT, useable only to reverse a boolean value

4. The following conditional and looping constructs require the use of boolean values: if while for do ?:

5. No other data type can be assigned or cast to boolean, and boolean can not be assigned or cast to any other data type.

ASSIGNMENT

P311.java

```
public class P311 {
    public static void main (String[] args) {

        int i = 1;                //assignment in declaration
        int j;
        j = 2;           //assignment in procedural statement

        int a, b, c;
        a = b = c = 5;                              //cascade

        i += 4;         //compound assignment, same as i=i+4;
        System.out.println ("1. i=" + i);

        //i=5; j=2; a=5; b=5; c=5;
        a *= i + j;
        b = b * i + j;
        c = c * (i + j);
        System.out.println ("2. a="+a + ", b="+b + ", c="+c);
    }
}
```

Result, P311.java

```
1. i=5
2. a=35, b=27, c=35
```

===

1. The simple assignment operator is the = equal sign.

2. The value of an assignment is the value assigned, and has the data type of the variable assigned.

3. Compound assignment operators combine binary operations (operations that take two operands) with assignment, such as:
   +=   -=   *=   /=   %=

   a. It is error-prone to use the same variable on both sides of a compound assignment operator.

4. Initialization in declarations is performed when the declaration is executed.

OPTIONAL

5. Both operands of compound assignments must be basic data types, except if the left-hand side is String, the right-hand side can be any type.

6. Casting is required when assigning some basic types of variables to others. Casting is summarized on page 3.20.

OPTIONAL: AUTOMATIC TYPE CONVERSION, CASTING NUMERIC TYPES

1. The numeric types are byte, short, int, long, float, and double.

2. Automatic type conversion and casting are not needed when a value is assigned to a variable of the same data type.

3. Widening and promotion are done automatically when numeric types differ:

   a. If the receiving variable is longer than the source, the source is automatically widened.

   b. If the receiving variable is a floating type and the source is an integer type, the integer is automatically promoted to floating type.

4. Widening of integers is done automatically in some cases:

   a. The values of types byte and short are widened to int before integer arithmetic is performed.

   b. Integer arithmetic is performed using int, EXCEPT if one or both operands are long, the shorter value is widened to long and the arithmetic is done in long.

5. Any integer type can be cast to any other integer type. You MUST cast if you assign a longer value to a shorter variable, because significant data could be lost. Casting is required in these cases:

   a. assigning to byte from short, int, or long
   b. assigning to short from int or long
   c. assigning to int from long

6. If the receiving variable is shorter, or integer when the source is floating point, there is danger of losing significant data. In this case you must use the cast operator to request the conversion.

7. When a floating point value is converted to integer, the fraction is truncated. Also, the integer part of the number may be too large to fit; see the next paragraph.

8. When the integer value of the source is too large to fit into the receiving variable, the least significant bits of the source are retained and the most significant bits are truncated.

OPTIONAL: CASTING EXAMPLE

P313.java

```
public class P313 {
    public static void main (String[] args) {3
        short  s;
        int    i = 45000;
        double d = 25.5;
        //s = i;   //Incompatible type for =. Explicit cast
        //i = d;   //needed for int to short or double to int
        s = (short) i;
        System.out.println ("i=" + i + ", s=" + s);
        i = (int) d;
        System.out.println ("d=" + d + ", i=" + i);
    }
}
```

Result, P313.java

```
i=45000, s=-20536
d=25.5, i=25
```

===

1. The cast operator consists of the name of a data type in ( ) parentheses preceding a variable or expression. Casting causes the value of the casted variable or expression to be treated as the cast data type for that one use of the value.

OPTIONAL, for those who know binary

```
public class P313a { public static void main(String[] args) {
  int   i = 45000;
  short s = (short) i;
  System.out.print ("i=" + i + ", s=" + s);
  String binaryI = Integer.toBinaryString(i);
  String binaryS = Integer.toBinaryString(s);
  String binaryP = Integer.toBinaryString(s * -1);
  System.out.println (
  "\nbinaryI=0000000000000000" + binaryI +
  "\nbinaryS=" + binaryS +
  "\nbinaryP reverse bits + 1=" + binaryP +
  "\nruler line               fedcba987654321" +
  "\nbinary place values:" +
  "\n16384 8192 4096 2048 1024 512 256 128 64 32 16 8 4 2 1"+
  "\nf     e    d    c    b    a   9   8   7  6  5  4 3 2 1"+
  "\n8+16+32+4096+16384 = " + (8 + 16 + 32 + 4096 + 16384) );
}  }
```

Result, P313a.java

```
i=45000, s=-20536
binaryI=00000000000000001010111111001000
binaryS=11111111111111111010111111001000
binaryP reverse bits + 1=101000000111000
ruler line               fedcba987654321
binary place values:
16384 8192 4096 2048 1024 512 256 128 64 32 16 8 4 2 1
f     e    d    c    b    a   9   8   7  6  5  4 3 2 1
8+16+32+4096+16384 = 20536
```

final VARIABLES

P314.java

```
public class P314 {
    public static void main (String[] args) {

        final double FINAL1 = 1.5;
        final double FINAL2;

        FINAL2 = 2.5;
        //FINAL2 = 3.5;

        System.out.println ("1=" + FINAL1 + ", 2=" + FINAL2);
    }
}
```

Result, P314.java

```
1=1.5, 2=2.5
```

1. A variable that is declared final becomes a named constant, and cannot be assigned a value more than one time.

2. The value for a final variable can be assigned in the declaration or in a procedural statement.

3. By convention, names of final variables are all uppercase.

EXERCISES

1. Create a program called E31.java to create two int variables called i1 and i2. Initialize i1 in the declaration statement. Initialize i2 in a procedural statement. Add the values of the two variables and display the total.

2. Create a program called E32.java. This program is the beginning of the case study for this course, a room reservation application for a training center.

   a. Variable declarations:

   ```
   int reservationNumber = 130323;
   int seats = 12;
   int numberOfDays = 5;
   double dayRatePerSeat = 25.00;
   double taxRate = 0.0725;
   ```

   b. Calculations:

roomAmount	product of seats, numberOfDays, and dayRatePerSeat
taxAmount	product of roomAmount and taxRate
finalAmount	sum of roomAmount and taxAmount

   c. Print your results. The output of the solution is as follows, but you may format your report differently.

   ```
   Reservation: 130323
   Number of seats: 12
   Number of days: 5
   Room Amount: 1500.0
   Tax at 7.249999999999999%: 108.74999999999999
   Final Amount: 1608.75
   ```

3. Copy your program E31.java and call the copy E33.java. ONE AT A TIME, make errors in the program as listed below and on the next page. Try to compile. If the program compiles try to execute. Correct each error before going on to the next.

   a. Declare two variables with the same name:

   ```
   int i;
   int i;
   ```

   b. Increment a variable that is not initialized.

   ```
   int i;
   i++;
   ```

c. Use a keyword as a variable identifier.

```
int new;
```

d. Declare a char variable with an initial value coded in double quotes (double quotes signify a String value).

```
char c = "c";
```

e. Assign a String to a char variable.

```
char c;
c = "c";
```

f. Assign an int to a boolean.

```
int i=1;
boolean b = true;
b = i;
```

g. Assign a boolean to an int.

```
int i=1;
boolean b = true;
i = b;
```

h. Optional: Assign a new value to a final variable that already has a value.

```
final int i=5;
i=6;
```

i. Optional: Assign an int to a boolean with casting.

```
int i=1;
boolean b = true;
b = (boolean) i;
```

j. Optional: Assign a boolean to an int with casting.

```
int i=1;
boolean b = true;
i = (int) b;
```

4. Copy your program E32.java and call the new copy E34.java. Use E34.java to experiment with the Eclipse features described on page 3.18.

SOLUTIONS

E31.java

```
public class E31 {
    public static void main (String[] args) {
        int i1 = 34;
        int i2;
        i2 = 567;
        i1 = i1 + i2;
        System.out.println ("total=" + i1);
    }
}
```

Result, E31.java

```
total=601
```

E32.java

```
public class E32 {
    public static void main (String[] args) {

        int reservationNumber = 130323;
        int seats = 12;
        int numberOfDays = 5;
        double dayRatePerSeat = 25.00;
        double taxRate = 0.0725;

        double roomAmount =
            seats * numberOfDays * dayRatePerSeat;
        double taxAmount = roomAmount * taxRate;
        double finalAmount =
            roomAmount + taxAmount;

        System.out.println (
           "Reservation: " + reservationNumber +
         "\nNumber of seats: " + seats +
         "\nNumber of days: " + numberOfDays +
         "\nRoom Amount: " + roomAmount +
         "\nTax at " + taxRate*100 + "%: " + taxAmount +
         "\nFinal Amount: " + finalAmount);
    }
}
```

Result, E32.java

```
Reservation: 130323
Number of seats: 12
Number of days: 5
Room Amount: 1500.0
Tax at 7.249999999999999%: 108.74999999999999
Final Amount: 1608.75
```

ECLIPSE

1. Comment out a section of code

    a. Highlight the code to be commented out.
        1) For /* */ click Source, Add Block Comment.
        2) For // on each line click Source, Toggle Comment.

2. Indent lines

    a. Indent: Highlight lines to be indented. Click Source, Shift Right.
    b. Unindent: Highlight lines to be unindented. Click Source, Shift Left.

3. Format your code

    a. Click Source, Format
    b. If some code is highlighted, only that part will be formatted. If no code is highlighted, the entire file will be formatted.

4. Maximize and reduce the Editor view

    a. Maximize: Double-click on the classname in the tab.
    b. Restore smaller size: Double-click on classname again.
    c. Toggle the Editor size and the display of other views: Click the tiny one- or two-rectangle buttons in the gray column next to the Editor's top left corner.

5. * in classname tab means the class in focus in the Editor has not been saved. To save it, click File, Save. Alternatively, click the floppy disk in the icon bar, or enter control s.

6. Problem icons in left margin bar

    a. Clipboard with blue checkmark: indicates the comment "// TODO Auto-generated method stub" and means that the method header and body is an auto-generated method stub.
    b. Yellow light bulb: warning.
    c. Red circle with white X: The current or next line has an error that is underlined in red. Hover your mouse on the red circle for an explanation. Hover over the underlined text for the explanation and a menu of fixes.

7. Text lines too long to display invoke a scroll-bar

    a. A scroll-bar will appear at the bottom to enable you to scroll right and left to see the entire line. Limit your line length to prevent the need for scrolling.

BASIC DATA TYPES, SUMMARY

1. Eight Basic Data Types:
   a. integers: byte, short, int, long
   b. floating-point: float, double
   c. character: char
   d. boolean: boolean

2. Integer Variables:

name	bits	bytes	signed?	range
byte	8	1	yes	-128 to 127
short	16	2	yes	-32,768 to 32,767
int	32	4	yes	-2,147,483,648 to 2,147,483,647
long	64	8	yes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

3. Integer Literals (default is int; L or l is long):
   a. decimal: 123 123L -123
   b. octal: 0123 0123L
   c. hexadecimal: 0x123 0x123L

4. Integer Operations (Bitwise are not covered in this course):
   a. Comparison < <= >= > == !=
   b. Arithmetic * / % + -
   c. Increment and decrement ++ --

5. Floating Point Variables:

name	bits	bytes	signed?	range
float	32	4	yes	1.40239846e-45 to 3.40282347e+38
double	64	8	yes	4.94065645841246544e-324 to 1.79769313486231570e+308

6. Floating Point Literals (default is double; F or f is float):
   a. float: 12.3f 45.67F
   b. double: 12.3 45.67 8.9E3

7. Floating Point Operations:
   a. Comparison < <= >= > == !=
   b. Arithmetic * / % + -
   c. Increment and decrement ++ --

8. char Literals (char is two bytes):
   'A' or '\101' or '\u0041' for the character A
   \' single quote \\ backslash \f formfeed \r return
   \" double quote \b backspace \n newline \t tab

9. Boolean Literals: true false

10. Boolean Operations:
   a. == != Equal to, Not equal to
   b. && || Short-circuiting logical operations AND, OR
   c. ! NOT, useable only to reverse a boolean value

11. Assignments:

```
a.  int i, j=2;      //assignment in declaration statement
b.  i = j;           //assignment in procedural statement
c.  i += 3;          //same as i=i+3; see +=  -=  *=  /=  %=
d.  a = b = c = 5;                               //cascade
e.  a *= i + j;                                  //same as g.
f.  a = a * i + j;
g.  a = a * (i + j);
```

12. Automatic Type Conversion and Widening of Numeric Types: byte, short, int, long, float, double.

    a. Automatic type conversion and casting are not needed when a value is assigned to a variable of the same data type.

    b. If the receiving variable is longer than the source, the source value is automatically widened.

    c. If the receiving variable is a floating type and the source is an integer type, the integer value is automatically promoted to floating type.

    d. The values of types byte and short are widened to int before integer arithmetic is performed.

    e. Integer arithmetic is performed using int, EXCEPT if one or both operands are long, the shorter value is widened to long and the arithmetic is done in long.

13. Casting Numeric Types:

    a. The cast operator consists of a data type name in () parentheses preceding a variable or expression. Casting causes the variable or expression's value to be treated as the cast type, but only for that one use of the value.
    Example: `double d = 12.34;    int i = (int) d;`

    b. If the receiving variable is shorter, or integer when the source is float or double, the cast operator is required because significant data might be lost.

    c. If a float or double value is cast and assigned to an integer type variable, the fraction is truncated.

    d. If a source integer value is too large to fit into the receiving variable, the LEAST significant bits are kept and the most significant bits are truncated.

14. You can cast int literals to long with l or L: 123L
    You can cast double literals to float with f or F: 1.23F

15. Casting is required if you read byte-oriented files in which the bytes must be interpreted as characters.
    `byte b = inputByte;  char c = (char) b;`

UNIT 4: CONDITIONAL AND LOOPING CONTROL STRUCTURES

Upon completion of this unit, students should be able to:

1. Use the flow of control constructs if, switch, while, do, for, break, and continue.

2. Use the ternary operator to select one of two expressions.

STATEMENTS AND BLOCKS, SCOPE

P402.java

```
1   public class P402 {
2       public static void main (String[] args) {
3           boolean mon = true;
4           boolean tue = false;
5
6           if (mon == true) {
7               boolean tue = true;
8               boolean wed = true;
9               System.out.println ("mon=" + mon
10                  + ", tue=" + tue + ", wed=" + wed);
11          }
12          //System.out.println ("mon=" + mon
13          //    + ", tue=" + tue + ", wed=" + wed);
14      }
15  }
```

Result, compile error in P402.java

```
P402.java:7: tue is already defined in main(java.lang.String[])
        boolean tue = true;
                ^
1 error
```

===

1. A statement can be:
    a. A simple statement ending in ; semicolon, or
    b. A compound statement ("block") within { } curly braces.

2. A block can contain zero or more other statements. Class and method bodies are created with the use of a block.

3. "Scope" means the block or section of code where an identifier exists and is recognized by the compiler.

4. Variables declared within a method are local to the method block, have scope from the point of declaration to the end of the block, and contain "garbage" (unpredictable bit settings) until assigned a value by your code.

5. Variables declared in an inner block inside a method are not accessible outside their block.

6. In the program above, if you comment out line 7 and uncomment lines 12 and 13, the compiler gives the error message:

```
P402.java:13: cannot find symbol
symbol  : variable wed
location: class P402
           + ", tue=" + tue + ", wed=" + wed);
                                         ^
1 error
```

IF AND COMPARISONS

P403.java

```
public class P403 {
    public static void main (String[] args) {
        int seats=10, days=2;

        if ((seats < 15) && (days == 1 || days == 2)) {
            System.out.println ("reservation accepted");
        } else {
            System.out.println ("must reschedule next week");
        }
    }
}
```

Result, P403.java

```
reservation accepted
```

1.

```
if (boolean_expression)
    true_statement;      //code convention is to use curlies
else
    false_statement;
next_statement;
```

2. The boolean_expression must be enclosed in parentheses, and must produce a boolean value.

3. If the boolean_expression is true, the true_statement is executed and then flow of control goes to the next_statement.

4. The else and false_statement are optional. If they are coded and the boolean_expression is false, the false_statement is executed and then flow of control goes to the next_statement.

5. The true_statement and false_statement can be simple statements ending in ; or blocks enclosed in { }.

6. Each else clause matches the immediately preceding as-yet unmatched if clause. Use { } blocks to make the logic clear.

7. Comparison operators that produce boolean results are:

<	less than	==	equal to
>	greater than	!=	not equal to
<=	less or equal	&&	short-circuiting AND
>=	greater or equal	||	short-circuiting OR

8. If the compiler can determine that a statement will never be executed due to program logic (an "unreachable" statement), you get an error.

SWITCH

P404.java

```java
public class P404 {
    public static void main (String[] args) {

        int num=20;

        switch (num - 5) {
            default: System.out.print ("no match, ");
            case 15: System.out.print ("num=15, ");
            case 28: System.out.print ("num=28, break, ");
                     break;
            case 3: case 24: System.out.print ("3 or 24, ");
        }

        System.out.println ("after the switch");
    }
}
```

Result, P404.java

```
num=15, num=28, break, after the switch
```

===

1.
```
switch (byte_char_short_or_int_expr) {     //curlies required
    case value1: statement1;
    case value2:                       //As of Java 7, data types
    case value3: statement2or3;        //include String, Enum, and
}                                      //the wrapper classes Byte,
next_statement;                        //Short, Integer, Character
```

2. The expression in parentheses can yield a byte, char, short, or int type. If the value is byte, char, or short, it is converted to int before searching is done.

3. Curly braces are required around the list of choices. Each case value must be a constant expression of the same data type as the expression in parentheses. Each case value must be unique (different from each other case value).

4. More than one case label may select the same entry point.

5. If no case constant matches, the default case is used. If no default is coded, flow of control goes to the next_statement. Only one default label may be used.

6. If a case value matches, all statements are performed in the order they are coded until a break, continue, return, or the end of the switch.

WHILE

P405.java

```
public class P405 {
    public static void main (String[] args) {

        //good style, test 1, process 1

        int i=1;                                        //initialize
        while (i < 3 ) {                                //test
            System.out.print (i + ", ");
            i++;                                        //increment
        }
        System.out.println("after, i=" + i);


        //bad style, test 0 but process 1

        int j=0;
        while (j < 3) {
            j++;
            System.out.print (j + ", ");
        }
        System.out.println("after, j=" + j);
    }
}
```

Result, P405.java

```
1, 2, after, i=3
1, 2, 3, after, j=3
```

===

1.
   ```
   while (boolean_expression)
       true_statement;       //code convention is to use curlies
   next_statement;
   ```

2. The boolean_expression must be enclosed in parentheses, and must produce a boolean value.

3. The true_statement can be a simple statement ending in ; or a block enclosed in { }.

4. If the boolean_expression is true, the true_statement is executed and then flow of control goes back to evaluation of the boolean_expression. Looping continues until the boolean_expression becomes false, and then flow of control goes to the next_statement. If the boolean_expression is false the first time it is evaluated, the true_statement is not executed.

DO

P406.java

```
public class P406 {
    public static void main (String[] args) {

        int i=0;                                   //initialize
        do {

            System.out.print (i + ", ");
            i++;                                   //increment

        } while (i < 3);                           //test
        System.out.println("after, i=" + i);
    }
}
```

Result, P406.java

```
0, 1, 2, after, i=3
```

1.
```
do
    true_statement;       //code convention is to use curlies
while (boolean_expression);   //unusual ; after )
next_statement;
```

2. The boolean_expression must be enclosed in parentheses, and must produce a boolean value.

3. The true_statement can be a simple statement ending in ; or a block enclosed in { }.

4. The true_statement is executed once and then if the boolean_expression is true, flow of control goes back to the true_statement. Looping continues until the boolean_expression becomes false, and then flow of control goes to the next_statement. If the boolean_expression is false the first time it is evaluated, the true_statement has already been executed once.

FOR

P407.java

```
public class P407 {
    public static void main (String[] args) {

        int i;
        for (i=0; i < 3 ; i++) {
            System.out.print (i + ", ");
        }
        System.out.println ("after: i=" + i);

        for (int j=0; j < 3 ; ++j)
            System.out.print (j + ", ");
        System.out.println ("after: j is not defined now");
    }
}
```

Result, P407.java

```
0, 1, 2, after: i=3
0, 1, 2, after: j is not defined now
```

1.
   ```
   for (initialization ; boolean_expr ; increment)
       true_statement;        //code convention is to use curlies
   next_statement;
   ```

2. The boolean_expr must yield a boolean value.

3. The parentheses must contain exactly two ; semicolons. The initialization is performed once when the loop is begun. Then the boolean_expr is evaluated; if true, the true_statement is executed and then flow of control goes to the increment, and then back to evaluation of the boolean_expr. Looping continues until the boolean_expr is tested and found to be false, and then flow of control goes to the next_statement. If the boolean_expr is false the first time it is evaluated, the true_statement is not executed.

4. The true_statement can be a simple statement ending in ; or a block enclosed in { }.

5. Each piece of code in parentheses can be omitted. If the boolean_expr is omitted, it defaults to true.

6. Multiple expressions in the initialization and increment must be separated by commas: for (i=0,j=1; i<4 && j<44; i++,j=j+4)

7. Variables declared in the initialization are local to the loop.

BREAK

P408.java

```
1   public class P408 {
2       public static void main (String[] args) {
3           int i=0, j=0;
4
5   /*1*/   while (i < 5) {
6               if (i == 3) break;              //go to line 10
7               System.out.print (i + ",  ");
8               i++;
9           }
10          System.out.println ("end 1: i=" + i + "\n");
11
12  /*2*/   for (i=0; i < 3; i++) {
13              for (j=0; j < 4; j++) {
14                  System.out.print (i + "" + j + ", ");
15                  if (i==1 && j==1) break;    //go to line 17
16              }
17              System.out.println ();
18          }
19          System.out.println ("end 2: i=" + i + ", j=" + j);
20      }
21  }
```

Result, P408.java

```
0,  1,  2,  end 1: i=3

00, 01, 02, 03,
10, 11,
20, 21, 22, 23,
end 2: i=3, j=4
```

1. The break statement changes the flow of control to the next_statement immediately after the end of the innermost enclosing switch, while, do, or for construct.

2. It is an error if break is not in a switch, loop construct, or labeled block. Labeled blocks are covered later in this unit.

3. To break out of nested loops, use a labeled block.

CONTINUE

P409.java

```
1   public class P409 {
2       public static void main (String[] args) {
3           int i=0, j=0;
4
5   /*1*/   while (i < 7) {
6               if ( (i%2) == 1) {
7                   //procedure for odd-numbered values of i
8                   i = i + 1;
9                   continue;                   //go to line 5
10              }
11              //procedure for even-numbered values of i
12              System.out.print (i + ", ");
13              i = i + 2;
14          }
15          System.out.println ("end 1: i=" + i + "\n");
16
17  /*2*/   for (i=0; i<3; i++) {
18              for (j=0; j<4; j++) {
19                  if (i==1 && j==1) continue;  //go to j++
20                  System.out.print (i + "" + j + ", ");
21              }
22              System.out.println ();
23          }
24          System.out.println ("end 2: i=" + i + ", j=" + j);
25      }
26  }
```

Result, P409.java

```
0, 2, 4, 6, end 1: i=8

00, 01, 02, 03,
10, 12, 13,
20, 21, 22, 23,
end 2: i=3, j=4
```

===

1. The continue statement changes the flow of control of the innermost enclosing while, do, or for loop as follows:

```
while    to the next evaluation of the boolean_expression
do       to the next evaluation of the boolean_expression
for      to the next evaluation of the increment
```

2. It is an error if continue is not within a loop construct.

3. To continue to an outer loop, use a labeled block (see next page).

LABELED BLOCKS WITH BREAK AND CONTINUE

P410.java

```
1   public class P410 {
2       public static void main (String[] args) {
3           int i=0, j=0;
4
5   /*1*/   i_loop: for (i=0; i < 3; i++) {
6               for (j=0; j < 4; j++) {
7                   System.out.print (i + "" + j + ", ");
8                   if (i==1 && j==1) break i_loop;  //to line 12
9               }
10              System.out.println ();
11          }
12          System.out.println ("end 1: i=" + i + ", j=" + j);
13
14  /*2*/   OUTER:
15          for (i=6; i < 8; i++) {
16              for (j=0; j < 4; j++) {
17                  System.out.print (i + "" + j + ", ");
18                  if (i==7 && j==1) continue OUTER; //go to i++
19              }
20              System.out.println ();
21          }
22          System.out.println ("end 2: i=" + i + ", j=" + j);
23      }
24  }
```

Result, P410.java

```
00, 01, 02, 03,
10, 11, end 1: i=1, j=1
60, 61, 62, 63,
70, 71, end 2: i=8, j=1
```

1. You can code a label on any statement.

2. The break can jump out of nested loops, or out of a labeled block, if coded with the label of a target loop or switch.

3. The continue can continue an outer loop, if coded with the label of a target loop.

OPTIONAL

4. A label on an arbitrary set of curly braces, together with a break statement, provide a form of goto:

```
        TOP: {  //procedure 1
                if (Boolean_expression) break TOP;
                //procedure 2
             }
```

TERNARY OPERATOR ? :

P411.java

```
public class P411 {
    public static void main (String[] args) {
        int i=1, j=2, k;

/*1*/   if (i==1) {
            k = i;
        } else {
            k = j;
        }

/*2*/   k = i==1 ? i : j;

/*3*/   k = (i==1 ? i : j);

        System.out.println ("1. k=" + k);

        System.out.println (i==j ? "2. i==j" : "2. i!=j");

        //j+9;                  //invalid expression statement
        //i==j ? j=9 : j=10 ;  //invalid expression statement
    }
}
```

Result, P411.java

```
1. k=1
2. i!=j
```

==

1. The ternary operator is a conditional operator that is used as a short-hand "if" to select one of two expressions.

2. The ternary operator has three operands separated by ? question mark and : colon.

```
boolean_expression ? true_expression : false_expression
```

3. If the boolean_expression is true, the true_expression is evaluated and its result is the value of the entire expression. If the boolean_expression is false, the false_expression is evaluated and its result is the value of the entire expression.

4. The true_expression and false_expression cannot be void.

EXERCISES

1. Create a program called E41.java that contains three loops, while, do, and for.

    a. The while loop generates and displays a series of numbers from 0 through 9.

    b. The do loop generates and displays a series of numbers from 10 through 19.

    c. The for loop generates and displays a series of numbers from 20 through 29.

2. Copy E32.java and call the copy E42.java. In E42.java keep your calculations and printout the same, but validate the values in the variables as follows. Error messages should be printed to the console via System.err rather than System.out.

    a. Validate seats via a switch structure. Valid values are 8, 10, 12, and 14. For invalid values, print an error message to the console, and set the seats to 12.

    b. Validate numberOfDays via an if structure. Valid values are 1 through 5. For invalid values, print an error message to the console, and set the numberOfDays to 5.

    c. Validate dayRatePerSeat via an if structure. Valid values are 25.00 through 65.00. For invalid values, print an error message to the console, and set dayRatePerSeat to 25.00.

    d. Validate taxRate via an if structure. Valid values are 0.05 through 0.20 For invalid values, print an error message to the console, and set taxRate to 0.0725.

    e. Execute your program several times, changing the value of each variable, to test your logic and make sure that each variable is correctly validated.

3. Copy your program E42.java and call the new copy E43.java. ONE AT A TIME, make errors in the program as listed below and try to compile it. If it compiles then try to execute it. Correct each error before going on to the next.

    a. In an if construct, use a capital I in the word If instead of lowercase i in the word if.

```
If (a == b) {
    //what to do;
}
```

b. In an if construct, use an integer expression for a condition, instead of a boolean expression. For example:

```
int a = 0;
if (a) {
    //what to do;
}
```

c. Code a ; semicolon after the condition of an if, for example:

```
if (i == j) ; {
    //what to do;
}
```

d. In an if construct, use a single equal sign (which means assignment) instead of a double equal sign (which is the equality comparison operator). In other words, use

```
if (a = b)
```

instead of

```
if (a == b)
```

e. Omit the closing curly brace of a block with the while loop.

f. Omit the opening curly brace of a block with the for loop.

g. Omit the : colon following the case value in the switch construct.

h. Code a floating point value in the parentheses of the switch construct.

4. Use your program E43.java to experiment with the Eclipse features described on page 4.14.

ECLIPSE

1. Content Assist

   a. switch: Type sw and press Control-Space. A popup will show your choices. Double-click to select your switch.

   b. if: Type if and press Control-Space. A popup will show your choices. Double-click to select your if.

   c. System.err.println(): Type syserr, press Control-Space.

2. Surround with if, while, do, for

   a. Highlight the code to be surrounded with a structure.
   b. Click Source, Surround With, and select the structure.

3. Highlight all occurrences of an identifier

   a. Rest your cursor on any identifier, or click on the identifier, and all occurrences where it is used in your code will be highlighted.

4. Highlight from open to close { } or [ ] or ( )

   a. Double-click on the character-position after (to the right of) an open { or [or (, and the editor highlights to the matching close } or] or).

   b. In some versions of Eclipse, including Mars, you can highlight from the end to the beginning via double-click on the character-position before (to the left of) a close } or] or).

5. Highlight and copy a line in the editor

   a. While your cursor is anywhere in the line: press the HOME key on your keyboard to move the cursor to the first nonblank character of that line. Press HOME again to go to column 1. Then press shift-downArrow to highlight the entire line.

   b. Then press Control-C, Control-V, Control-V. Your first Control-V makes the line overwrite itself and your second Control-V makes a new copy of the line below the original line.

SOLUTIONS

E41.java

```
public class E41 {
    public static void main (String[] args) {
        int i=0;

/*a*/   while (i < 10) {
            System.out.print (i + ", ");
            i++;
        }
        System.out.println ("\n");

/*b*/   do {
            System.out.print (i + ", ");
            ++i;
        } while (i < 20);
        System.out.println ("\n");

/*c*/   for ( ; i<30 ; i++) {
            System.out.print (i + ", ");
        }
        System.out.println ("\n");
    }
}
```

Result, E41.java

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,

10, 11, 12, 13, 14, 15, 16, 17, 18, 19,

20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
```

E42.java

```
public class E42 {
    public static void main (String[] args) {

        int reservationNumber = 130323;
        int seats = 2;                          //invalid value
        int numberOfDays = 6;                   //invalid value
        double dayRatePerSeat = 75.00;          //invalid value
        double taxRate = 0.2025;                //invalid value

        switch (seats) {
            case 8:  break;
            case 10: break;
            case 12: break;
            case 14: break;
            default: System.err.println ("Invalid seats "
                         + seats + ", will be set to 12");
                     seats = 12;
        }

        if (numberOfDays < 1 || numberOfDays > 5) {
            System.err.println ("Invalid numberOfDays "
                + numberOfDays + ", will be set to 5");
            numberOfDays = 5;
        }

        if (dayRatePerSeat<25.00 || dayRatePerSeat>65.00) {
            System.err.println ("Invalid dayRatePerSeat "
                + dayRatePerSeat + ", will be set to 25.00");
            dayRatePerSeat = 25.00;
        }

        if (taxRate < 0.05 || taxRate > 0.20) {
            System.err.println ("Invalid taxRate "
                + taxRate + ", will be set to 0.0725");
            taxRate = 0.0725;
        }

        double roomAmount =
            seats * numberOfDays * dayRatePerSeat;
        double taxAmount = roomAmount * taxRate;
        double finalAmount =
            roomAmount + taxAmount;

        System.out.println (
           "Reservation: " + reservationNumber +
         "\nNumber of seats: " + seats +
         "\nNumber of days: " + numberOfDays +
         "\nRoom Amount: " + roomAmount +
         "\nTax at " + taxRate*100 + "%: " + taxAmount +
         "\nFinal Amount: " + finalAmount);
    }
}
```

Result, E42.java

```
Invalid seats 2, will be set to 12
Invalid numberOfDays 6, will be set to 5
Invalid dayRatePerSeat 75.0, will be set to 25.00
Invalid taxRate 0.2025, will be set to 0.0725
Reservation: 130323
Number of seats: 12
Number of days: 5
Room Amount: 1500.0
Tax at 7.249999999999999%: 108.74999999999999
Final Amount: 1608.75
```

OPTIONAL NUMBER FORMATTING

To format dollar amounts and percentages so they look appropriate when printed, these lines can be used. Do not forget the import line above the class header. Number formatting is covered in the optional Unit 17.

E42FormattedNumbers.java, Excerpts

```
import java.text.NumberFormat;        //needed for formatting

public class E42FormattedNumbers {
~~~~
        NumberFormat money =
            NumberFormat.getCurrencyInstance ();
        String moneyRoomAmount = money.format(roomAmount);
        String moneyTaxAmount = money.format(taxAmount);
        String moneyFinalAmount = money.format(finalAmount);

        NumberFormat x100 = NumberFormat.getInstance ();
        String x100TaxRate = x100.format(taxRate * 100);

        System.out.println (
           "Reservation: " + reservationNumber +
         "\nNumber of seats: " + seats +
         "\nNumber of days: " + numberOfDays +
         "\nRoom Amount: " + moneyRoomAmount +
         "\nTax at " + x100TaxRate + "%: " + moneyTaxAmount +
         "\nFinal Amount: " + moneyFinalAmount);
    }
}
```

Result, E42FormattedNumbers.java

```
Invalid seats 2, will be set to 12
Invalid numberOfDays 6, will be set to 5
Invalid dayRatePerSeat 75.0, will be set to 25.00
Invalid taxRate 0.2025, will be set to 0.0725
Reservation: 130323
Number of seats: 12
Number of days: 5
Room Amount: $1,500.00
Tax at 7.25%: $108.75
Final Amount: $1,608.75
```

(blank)

UNIT 5: METHOD CALLS

Upon completion of this unit, students should be able to:

1. Code a class with more than one method.

2. Pass arguments to a method, and obtain the return value.

3. Briefly explain how overloading works, and code a class with overloaded methods.

METHODS AND THE STACK

1. A class can contain many methods and variables. These parts of a class are called its members.

2. A method can contain local variables, which are not members of the class. Local variables include:

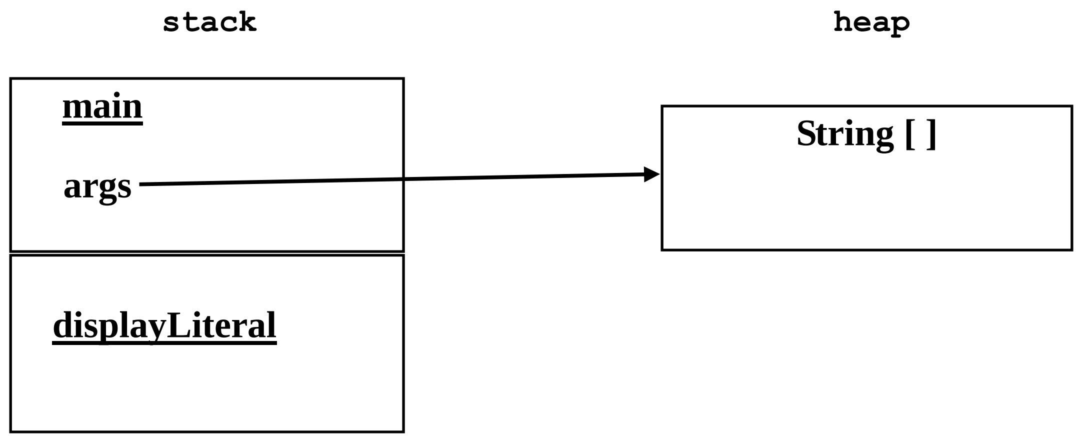
   a. Parameters received by the method, and
   b. Variables declared within the curlies of the method body.

3. During program execution, the JVM uses a stack area to hold executing methods with their local variables. (Another area called the heap is used for objects, to be covered in the next unit.)

4. Space in the stack for an executing method is allocated when the method is called, and deallocated when the method returns. After a method's stack area is deallocated, its space can be used by the next method to be called.

5. The method main occupies the first stack area, and its stack area is present during the entire program execution. When main returns to the JVM, the program stops executing.

6. The stack and heap for P503.java



A CLASS WITH TWO METHODS

P503.java

```java
public class P503 {

    public static void main (String[] args) {
        System.out.println ("1. main method before");
        displayLiteral();
        System.out.println ("3. main method after");
    }

    public static void displayLiteral () {
        System.out.println ("2. displayLiteral method");
    }
}
```

Result, P503.java

```
1. main method before
2. displayLiteral method
3. main method after
```

---

1. A method is a named, callable piece of code. All methods must be within a class.

2. The parts of a method include:

    a. Header.

        1) Optional modifiers, such as public or static, to be discussed in later units.

        2) The data type of the return value, or void if the method does not return a value.

        3) Identifier of the method.

        4) Parentheses enclosing the list of parameters to be received. Each parameter must be specified with its data type and its identifier to be used within the method.

    b. Body.

        1) Block enclosing the statements of the method.

3. In the program above, main is the calling method, and displayLiteral is the called method.

ARGUMENTS AND PARAMETERS

P504.java

```
public class P504 {
    public static void main (String[] args) {
        int i=1, j=23;

        System.out.println ("main before: " + i + " " + j);
        displayParams (i, j, 456);
        System.out.println ("main after: " + i + " " + j);
    }

    public static void displayParams (int a, int b, int c) {
        System.out.println ("1. " + a + " " + b + " " + c);
        a = 100;
        b = 200;
        c = 300;
        System.out.println ("2. " + a + " " + b + " " + c);
    }
}
```

Result, P504.java

```
main before: 1 23
1. 1 23 456
2. 100 200 300
main after: 1 23
```

==

1. The values passed to a method are called arguments.

2. The copies of the arguments that are received by the called method are called parameters.

3. Each parameter received is stored in a variable that is local to the called method.

4. Parameters are copies of the arguments passed. Changes to parameter variables in the called method do NOT affect the original values in the calling method.

5. The parameter list in parentheses must specify, for each parameter:

   a. Its data type
   b. The identifier to be used for it within the called method.

6. Parentheses are required, even if no parameters are coded.

return STATEMENT, RETURN VALUE, MULTIPLE POINTS OF RETURN

P505.java

```java
public class P505 {
    public static void main (String[] args) {
        int returnValue;

        returnValue = compareTwo (5, 5);
        myPrint (returnValue);

        myPrint (compareTwo (6,7));  //arguments are resolved
    }                                //before the method call
    public static int compareTwo (int n1, int n2) {
        if (n1 == n2)
            return 0;
        else
            return -1;
    }
    public static void myPrint (int n) {
        if (n == 0) {
            System.out.println ("Matching numbers");
            return;
        }
        System.out.println ("Non-matching numbers");
    }
}
```

Result, P505.java

```
Matching numbers
Non-matching numbers
```

==

1. A called method stops executing and program control returns to the calling method in two ways:

    a. Execution reaches the closing curly brace.

    b. A return statement is executed. Return statements can be coded with or without a return value.

2. If a method header specifies a return type other than void, the method must return via a return statement, and the return statement must return a value.

3. The return value must be able to be assigned to a variable of the type specified in the header without casting.

4. A void method can have return statement(s), but they cannot return a value. A call to a void method cannot be used as a value in an expression.

SINGLE POINT OF RETURN, System.exit(exitValue)

P506.java

```
public class P506 {
    public static void main (String[] args) {
        int returnValue;

        returnValue = compareTwo (5, 5);
        myPrint (returnValue);

        myPrint (compareTwo (6,7));  //arguments are resolved
                                     //before the method call
        System.exit(0);
    }
    public static int compareTwo (int n1, int n2) {
        int returnValue = 99;
        if (n1 == n2)
            returnValue = 0;
        else
            returnValue = -1;
        return returnValue;
    }
    public static void myPrint (int n) {
        if (n == 0) {
            System.out.println ("Matching numbers");
        } else {
            if (n == -1) {
                System.out.println ("Non-matching numbers");
            } else {
                System.out.println ("Should never be used");
            }
        }
        return;
    }
}
```

Result, P506.java

```
Matching numbers
Non-matching numbers
```

===

1. There are two styles of coding return values. Methods on the previous page have multiple points of return. The methods above have a single point of return.

2. If main() returns, control goes back to the JVM and program execution ends.

3. Normally main does not end with return but with the method System.exit() which sends an exit number to the JVM.

4. Exit numbers must be integers. A value between 0-255 inclusive will be most platform-independent.

OVERLOADING, ALSO CALLED COMPILE-TIME POLYMORPHISM

P507.java

```java
public class P507 {
    public static void main (String[] args) {

        boolean ret1 = displayData ("String argument");
        boolean ret2 = displayData (5.7);

        System.out.println ("1=" + ret1 + ", 2=" + ret2);
    }

    public static boolean displayData (String s) {
        if (s == null) {
            return false;
        }
        System.out.println (s);
        return true;
    }

    public static boolean displayData (double d) {
        System.out.println ("double value is " + d);
        return true;
    }
}
```

Result, P507.java

```
String argument
double value is 5.7
1=true, 2=true
```

1. Overloading is the technique of giving two or more methods the same name and different parameter lists.

2. The return types of overloaded methods may be the same or different, but are usually the same.

3. The compiler calls the correct version of an overloaded method according to the arguments passed.

4. The purpose of overloading is to create an illusion of simplicity by enabling your code to call the "same" method with different arguments and achieve the "same" result.

    a. System.out.print and System.out.println are overloaded.

    b. The javadoc for the class PrintStream shows these two methods in the Method Summary part of the documentation.

5. When a parameter is a non-basic type, it should be validated to ensure that it is not null before it is processed.

OVERLOADED METHODS CAN CALL EACH OTHER

P508.java

```java
public class P508 {
    public static void main (String[] args) {
        boolean b1 = displayStartDay ("Monday");
        boolean b2 = displayStartDay (2);
        boolean b3 = displayStartDay ();
        System.out.println ("1="+b1+", 2="+b2+", 3="+b3);
    }
    public static boolean displayStartDay (String s) {
        if (s == null) {
            return false;
        }
        System.out.println ("Class starts on " + s);
        return true;
    }
    public static boolean displayStartDay (int day) {
        String p;
        switch (day) {
            case 1: p="Monday";    break;
            case 2: p="Tuesday";   break;
            case 3: p="Wednesday"; break;
            case 4: p="Thursday";  break;
            case 5: p="Friday";    break;
            default: return false;
        }
        boolean tmp = displayStartDay (p);
        return tmp;
        //same as: return displayStartDay(p);
    }
    public static boolean displayStartDay () {
        return false;
    }
}
```

Result, P508.java

```
Class starts on Monday
Class starts on Tuesday
1=true, 2=true, 3=false
```

===

1. One overloaded method can call another. This is a common practice to avoid duplication of code.

2. An overloaded method that receives no parameters often creates one or more default value(s) for another overloaded method.

EXERCISES

NOTE: All methods must be public and static.

1. Create a program called E51.java that is similar to E41.java. In addition to the main method, E51.java contains three methods called whileLoop, doLoop, and forLoop. Each method contains one loop of the type specified in the method name.

   a. The whileLoop method contains a while loop that generates and displays a series of numbers from 0 through 9.

   b. The doLoop method contains a do-while loop that generates and displays a series of numbers from 10 through 19.

   c. The forLoop method contains a for loop that generates and displays a series of numbers from 20 through 29,

2. Copy E42.java and call the copy E52.java. In E52.java create the five methods described below. Use the validation procedures you created in the previous unit.

   a. Create a method called validateSeats that receives an int parameter, validates it, and returns the int or the default value 12. In the main method, call validateSeats with an int, and assign the returned value to seats.

   b. Create a method called validateNumberOfDays that receives an int parameter, validates it, and returns the int or the default value 5. In the main method, call validateNumberOfDays with an int, and assign the returned value to numberOfDays.

   c. Create a method called validateDayRatePerSeat that receives a double parameter, validates it, and returns the double or the default value 25.00. In the main method call validateDayRatePerSeat with a double, and assign the returned value to dayRatePerSeat.

   d. Create a method called validateTaxRate that receives a double parameter, validates it, and returns the double or the default value 0.0725. In the main method, call validateTaxRate with a double, and assign the returned value to taxRate.

   e. Create a void method called printOneReservation to receive seven parameters, which are the reservation variables to be printed. This method should print them. In the main method first calculate the amounts, and then call printOneReservation with the values to be printed.

ECLIPSE

1. Move the Editor to a method header in the current class.

   a. In the Editor, hover the mouse on a methodname where your code calls the method, hold down the control key and click. The cursor moves to the method header.

2. Outline view, Move the Editor to an item in the Outline view

   a. The Outline view displays the names and categories of the members of the class currently in focus in the Editor.

   b. Click on an item in the Outline view to move the Editor to that code.

   c. Outline view icons:

      1) Filled vs unfilled (this may differ in different versions and derivatives of Eclipse)
         Methods: filled
         Variables: unfilled

      2) Icon shape shows member accessibility

Private	red square
Protected	yellow diamond
Public	green circle
Unspecified "package friendly"	blue triangle

      3) Letters

C	constructor
S	static
A	abstract
F	final

3. Find any text: Click Edit, Find/Replace, type the text you want to find, click Find.

4. Move the cursor to another series of characters that are the same as highlighted text:

   a. Highlight the text to be found, such as an identifier.
   b. Press control-k. Or you can click Edit, Find Next.

5. Display 2 files in side-by-side Editors

   a. You can display and work with two or more classes side by side or above/below each other.

   b. Open both classes. Hold down the left mouse button on the file's Editor tab and drag-and-drop it to the desired location. As you are dragging the tab, the Editor will display a rectanglular gray outline where the Editor thinks you want the file to be displayed.

SOLUTIONS

E51.java

```
public class E51 {
    public static void main (String[] args) {
        whileLoop();
        doLoop();
        forLoop();
    }

    public static void whileLoop () {
        int i=0;
        while (i < 10) {
            System.out.print (i + ", ");
            i++;
        }
        System.out.println ();
    }

    public static void doLoop () {
        int i=10;
        do {
            System.out.print (i + ", ");
            i++;
        } while (i < 20);
        System.out.println ();
    }

    public static void forLoop () {
        for (int i=20; i<30 ; i++)
            System.out.print (i + ", ");
        System.out.println ("\n");
    }
}
```

Result, E51.java

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
```

E52.java

```
public class E52 {
    public static void main (String[] args) {

        int reservationNumber = 130323;
        int seats = validateSeats (12);
        int numberOfDays = validateNumberOfDays (5);
        double dayRatePerSeat =
            validateDayRatePerSeat (25.00);
        double taxRate = validateTaxRate (0.0725);

        double roomAmount =
            seats * numberOfDays * dayRatePerSeat;
        double taxAmount = roomAmount * taxRate;
        double finalAmount = roomAmount + taxAmount;

        printOneReservation (reservationNumber, seats,
            numberOfDays, roomAmount, taxRate, taxAmount,
            finalAmount);
    }

    public static int validateSeats (int s) {
        switch (s) {
            case 8:  break;
            case 10: break;
            case 12: break;
            case 14: break;
            default: System.err.println ("Invalid seats "
                         + s + ", will be set to 12");
                     return 12;
        }
        return s;
    }

    public static int validateNumberOfDays (int n) {
        if (n < 1 || n > 5) {
            System.err.println ("Invalid numberOfDays "
                + n + ", will be set to 5");
            return 5;
        }
        return n;
    }

    public static double validateDayRatePerSeat (double d) {
        if (d < 25.00 || d > 65.00) {
            System.err.println ("Invalid dayRatePerSeat "
                + d + ", will be set to 25.00");
            return 25.00;
        }
        return d;
    }

    public static double validateTaxRate (double t) {
        if (t < 0.05 || t > 0.20) {
            System.err.println ("Invalid taxRate "
                + t + ", will be set to 0.0725");
            return 0.0725;
        }
        return t;
    }

    public static void printOneReservation (
    int reservationNumber, int seats, int numberOfDays,
    double roomAmount, double taxRate, double taxAmount,
    double finalAmount) {
        System.out.println (
           "Reservation: " + reservationNumber +
         "\nNumber of seats: " + seats +
         "\nNumber of days: " + numberOfDays +
         "\nRoom Amount: " + roomAmount +
         "\nTax at " + taxRate*100 + "%: " + taxAmount +
         "\nFinal Amount: " + finalAmount);
    }
}
```

Result, E52.java

```
Reservation: 130323
Number of seats: 12
Number of days: 5
Room Amount: 1500.0
Tax at 7.249999999999999%: 108.74999999999999
Final Amount: 1608.75
```

(blank)

UNIT 6: CLASSES, OBJECTS, AND REFERENCES

Upon completion of this unit, students should be able to:

1. Briefly explain the difference between basic and reference data types.

2. Briefly explain the concept of encapsulation, and how it is implemented in Java.

3. Code and instantiate classes that make use of constructors, instance variables, and instance methods.

4. Pass a reference as an argument to a method; in the called method use the copy of the reference that has been received to modify variables in the object.

CLASSES, ENCAPSULATION

1. Java programs are organized into classes, and classes are used to modularize the program.

    a. All executable code (whether methods or variables) must be contained in a class.

    b. Most classes contain both variables and methods, which may be called data members and method members. Non-static members may be called instance variables and instance methods.

2. A class must be defined within a single source file, and cannot be separated into multiple files.

3. More than one class can be in one file, but this is not often done. Only one class in a file can be public. The name of the public class must be used as the filename, with the .java filename extension.

4. How are classes designed? Things of interest in the real-world problem domain are analyzed to determine their characteristics: what information do we know about them, and what do they do. Then, classes are designed to represent the aspects that are useful.

    a. Information is implemented in a class as variables, which may also be called data attributes or properties.

    b. What things do is implemented in a class as methods, which may be called the public interface of the class or its behavior, actions, or procedures.

5. Encapsulation is one benefit of object-oriented programming. Classes are designed to encapsulate their variables and procedures, and to provide a "public interface" that consists of public methods that other classes can call to make use of the encapsulated variables and procedures.

    a. An object created from a class definition should be a self-contained entity with a public interface, so that internal workings and data are private and can be changed without having to change how other program statements make use of the object.

    b. Typically, the variables of a class are private and the methods are public. The keywords private and public are discussed in a later unit.

CLASSES, OBJECTS, MORE TERMINOLOGY

1. A class is a unit of programming code. A class may be called a user-defined data type or just a type.

2. An object is an allocation of space made by the JVM during runtime to hold the variables and methods coded in a class.

    a. An object is an instance of a class, like an int variable is an instance of the basic data type int which is defined in the compiler.

    b. An object is also called an instance. Making an object is also called instantiation or creating an instance of the class or instantiating the class.

3. When your program creates an object, usually you will also create a reference variable that points to the object. When a reference is printed you may see:

    a. The class type that the object implements
    b. An @ at sign
    c. A hex number which serves as the symbolic address of the object in the program's heap space.

4. The JVM stores objects in a part of storage called the heap.
    a. An object's storage is allocated when the JVM executes the line(s) of programming code that create the object.
    b. When an object's reference count goes down to zero, the object is deallocated so its space can be used by other objects.

5. Storage deallocation is done by a process called gc, the garbage collector. gc runs at low priority. It looks for and deallocates objects with reference counts of zero.

6. Some object-oriented terminology:

    a. The variables in a class provide its attributes or "define the state" of an object of that class.

    b. The methods in a class define the "behavior" of an object of the class. Methods are called functions, subroutines, or procedures in some languages.

    c. When a method in an object of class One calls a method in an object of class Two, you can say that:

        1) The object of class One sent a message to the object of class Two.

        2) The object of class One invoked a method, or called a member function, in the object of class Two.

APPLICATIONS WITH MULTIPLE CLASSES, COMPILE AND EXECUTE

1. A stand-alone application must have a main class, sometimes called a driver class or test class, that contains the main method. To initiate execution of the application, you initiate execution of the JVM and give it the name of your main class.

2. Typically, in addition to the main class, an application will make use of many, sometimes hundreds, of business classes.

   a. Each business class provides one kind of functionality, such as attaching to an input source (such as a file, network line, or database) and reading from the source, or performing a business algorithm, etc.

   b. Business classes do not contain a main method, so they cannot execute as applications by themselves.

3. When an application contains many classes, two ways to compile on a commandline are:

   a. Specify the main class only. javac will compile the source files of every class used by your main class and other classes if the source files can be found, and if the source files have been modified more recently than their most recent bytecode files were created.

   ```
   $ javac P607.java                      ---UNIX commandline
   C:\myjava> javac P607.java             ---DOS commandline
   ```

   b. Specify the .java file(s) that you want compiled, even if their source code file(s) have not been changed since their most recent bytecode file(s) were created. You might do this to get the same timestamp on all bytecode files.

   ```
   $ javac P607.java T.java               ---UNIX commandline
   C:\myjava> javac P607.java T.java      ---DOS commandline
   ```

4. Each class is compiled into a separate .class bytecode file.

5. To execute an application containing many classes, invoke the JVM with the name of the main class.

   ```
   $ java P607                                ---UNIX commandline
   C:\myjava> java P607                       ---DOS commandline
   ```

6. In Eclipse, to compile all classes in an application so that all bytecode files have the current time as their timestamp, click Project, Clean. Click "Clean projects selected below". Select your projects to be cleaned. Click OK.

THE new OPERATOR AND CONSTRUCTORS

1. After a class has been defined, an object of its type can be created. Usually another class (called the driver class, main class, or test class) contains the code to create an object.

2. All objects are created by the new operator during runtime ("dynamically"). There are two steps that may be done in one statement or two.

   a. One statement: TCourse6 tc = new TCourse6 ("UNIX", 10);

   b. Two statements:

      i. Create a reference variable of the class type to point to the object. If defined in a method, the reference will contain the class type and garbage as the address of the object. If defined in an object in the heap, the address will be all zeroes.

      ii. Use new to create the object. This operator allocates the space in the heap for the object and returns a symbolic address (a hex number that is not a real byte number in RAM). You can assign this address to your reference variable.

```
TCourse6 tc;
tc = new TCourse6 ("Java", 12);
```

3. The classname() used with the new operator calls a constructor of that class. Constructors are similar to methods but a little different:

   a. Can have zero or more parameters in parentheses.
   b. Must have the same name as their class.
   c. Must not have a coded return type.
   d. Can be called only by the new operator when you create a new object.
   e. Usually initialize the instance variables of their object ("initialize the internal state of the object").

4. Heap space is cleared to all bits zero before an object is created. Variables in objects will contain all bits set to zero until they are set to other values.

   a. Variables with all bits zero are interpreted appropriately: numbers contain 0 or 0.0, chars contain '\u0000', booleans contain false; references contain null addresses.

5. The JVM's class loader loads a class the first time its name is used in a line of code being executed during runtime.

THE STACK AND HEAP, MEMORY ALLOCATION DURING P607

1. The JVM begins program execution by calling P607's main method, and allocating a stack area for main.

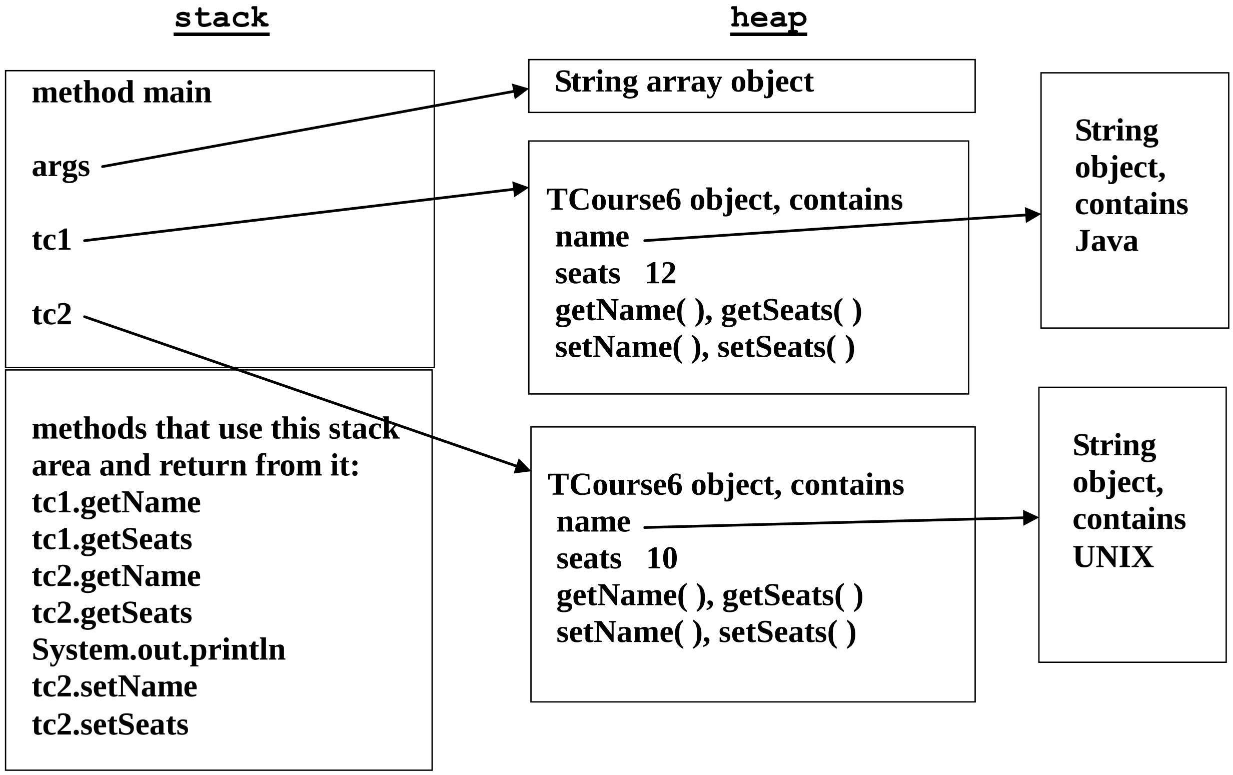
2. The variable args is created in main's stack area. It points to an array in the heap. Arrays are covered in a later unit.

3. The reference tc1 is created. Then the new operator creates a TCourse6 object in the heap, and returns the object's address which is assigned to tc1. The same steps are used to create the reference tc2 and the object it points to.

4. A stack area is created for tc1.getName(), and after it executes and returns, the stack area is deallocated. The same stack area is reallocated and deallocated for tc1.getSeats(), tc2.getName(), tc2.getSeats(), and System.out.println().

5. The set methods of tc2 are called to modify the data in the object. Then the five methods listed in 4. are called again.

6. The method main returns to the JVM. When main's stack area is deallocated all variables there are deallocated. When the reference variables in main are deallocated, the objects they reference are garbage collected because their reference counts go down to zero.



THE new OPERATOR AND CONSTRUCTORS, EXAMPLE

P607.java

```
1   public class P607 {                              //main class
2       public static void main (String[] args) {
3
4           TCourse6 tc1;
5           tc1 = new TCourse6 ("Java", 12);
6
7           TCourse6 tc2 = new TCourse6 ("UNIX", 10);
8
9           System.out.println("1. " + tc1 + ", " + tc2 + ",  " +
10              tc1.getName() + " has " + tc1.getSeats() + ", " +
11              tc2.getName() + " has " + tc2.getSeats() );
12
13          tc2.setName ("XML");
14          tc2.setSeats (14);
15
16          System.out.println("2. " + tc1 + ", " + tc2 + ",  " +
17              tc1.getName() + " has " + tc1.getSeats() + ", " +
18              tc2.getName() + " has " + tc2.getSeats() );
19      }
20  }
```

TCourse6.java

```
1   public class TCourse6 {                     //business class
2
3       private String name;                    //instance vars
4       private int seats;                      //will be in each
5                                               //object instance
6       public TCourse6 (String newName, int newSeats) {
7           setName (newName);                  //constructor
8           setSeats (newSeats);
9       }
10                                              //public instance
11      public String getName() {               //methods will be
12          return name;                        //in each object
13      }                                       //and provide the
14      public void setName(String newName){    //class's public
15          name = newName;                     //interface.
16      }                                       //The methods on
17      public int getSeats() {                 //lines 11-22 are
18          return seats;                       //called get and
19      }                                       //set methods, or
20      public void setSeats(int newSeats){     //getters and
21          seats = newSeats;                   //setters.
22      }
23  }
```

Result, P607.java

```
1. TCourse6@1845568, TCourse6@1032cf5,  Java has 12, UNIX has 10
2. TCourse6@1845568, TCourse6@1032cf5,  Java has 12, XML has 14
```

REFERENCES

1. When a reference is printed you may see
   a. The class type that the object implements.
   b. An @ at sign.
   c. A hex number which serves as the symbolic address of the object in the program's heap space.

2. No arithmetic can be done with the pointer value of a reference. References cannot be explicitly dereferenced.

3. If two references point to the same object, either reference can be used to access that object.

4. Every object has an implicit reference count variable.
   a. If the reference count of an object goes down to zero, the object is garbage collected.
   b. To deallocate an object, assign null to its reference. That decrements the object's reference count by one. If that causes the reference count to go down to zero, the object is garbage collected.

5. If a method receives a parameter that is a reference, the method should use an if to ensure that the reference is not null before using it. Dereferencing a null reference causes NullPointerException.

6. If two references point to objects of the same class type, one reference can be assigned to the other. Afterward both references point to the same object; the reference count of one object goes up by 1, and the other goes down by 1.
   a. After line 13 assigns the value in tc1 to tc2, the reference count of the object with Java,12 goes up to 2 and the reference count of the object with UNIX,10 goes down to zero. The reference count of zero causes the object with UNIX,10 to be garbage-collected.
   b. After tc2 is assigned null, the reference count of the object with Java,12 goes down to 1.

7. When would more than one object of the same class type be in the heap at the same time?
   a. Online purchase, the shopping cart holds many items.
   b. Warehouse picking and shipping lists for an order that has many items.
   c. Training class roster with many students.
   d. Investor portfolio with many funds.
   e. Insurance beneficiary list with multiple beneficiaries.
   f. Airline flight manifest with many passengers.
   g. Web page with multiple interactive components (buttons, checkboxes, text-entry areas) each of which requires a "handler" object to handle any interaction on the component.

REFERENCES, EXAMPLE

P609.java

```
public class P609 {
    public static void main (String[] args) {

        TCourse6 tc1 = new TCourse6 ("Java", 12);
        TCourse6 tc2 = new TCourse6 ("UNIX", 10);

        System.out.println ("1. " + tc1 + ", " + tc2);

        System.out.println ("2. " +
            tc1.getName() + " has " + tc1.getSeats() + ", " +
            tc2.getName() + " has " + tc2.getSeats() );

        tc2 = tc1;  //2 refs to the object with Java,12
                    //gc deallocates the object with UNIX,10

        System.out.println ("3. " + tc1 + ", " + tc2);

        System.out.println ("4. " +
            tc1.getName() + " has " + tc1.getSeats() + ", " +
            tc2.getName() + " has " + tc2.getSeats() );

        tc2 = null;    //ref count of Java,12 goes down to 1

        System.out.println ("5. " + tc1 + ", " + tc2);

        //tc2.setName("would cause NullPointerException");
    }
}
```

Result, P609.java

```
1. TCourse6@17c8f7f, TCourse6@b10190
2. Java has 12, UNIX has 10
3. TCourse6@17c8f7f, TCourse6@17c8f7f
4. Java has 12, Java has 12
5. TCourse6@17c8f7f, null
```

PASS A REFERENCE TO A CALLED METHOD

1. A reference variable points to an object, and does NOT contain the variables and values of the object.

2. When a reference is passed to a method, the method receives a copy of the reference. By using the copy, your called method can call methods in the object. If there are accessible get and/or set methods in the object, your called method can use them to access and/or change variables in the object.

```
      Stack                               Heap

___________________      ___________________     _______________
|                 |      |                 |     |             |
| main            |      | TCourse6 object |     |String object|
|                 |      |                 |     |             |
| roomsOpen  6    |      | name            |     |    Java     |
|                 |      | String@12cd---------->|             |
| tc1             |      |                 |     |             |
| TCourse6@b10190------->| seats 17        |     |_____________|
|                 |      |                 |
|                 |      | getName()       |
|_________________|      | setName()       |
| This stack area |      | getSeats()      |
| is occupied by  |      | setSeats()      |
| getName(),      |      |_________________|
| getSeats()      |
| and println()   |
| before and      |      ___________________     _______________
| after changeObj |      |                 |     |             |
|                 |      | TCourse6 object |     |String object|
| changObj        |      |                 |     |             |
|                 |      | name            |     |    UNIX     |
| rooms  123      |      | String@34ef---------->|             |
|                 |      |                 |     |             |
| t1              |      | seats 9         |     |_____________|
| TCourse6@ddc581------->|                 |
|_________________|      | getName()       |
                         | setName()       |
                         | getSeats()      |
                         | setSeats()      |
                         |_________________|
```

PASS A REFERENCE TO A CALLED METHOD, EXAMPLE

P611.java

```
public class P611 {
    public static void main (String[] args) {

        int roomsOpen = 6;
        TCourse6 tc1 = new TCourse6 ("Java", 12);

        System.out.println("1. roomsOpen=" + roomsOpen + ", "
         +tc1+ ", " + tc1.getName() + ", " + tc1.getSeats());

        if ( changeObj (roomsOpen, tc1) == true) {
        System.out.println ("5. roomsOpen=" + roomsOpen +", "
         +tc1+ ", " + tc1.getName() + ", " + tc1.getSeats());
        }
    }
    public static boolean changeObj(int rooms, TCourse6 t1) {

        if (t1 == null) { return false; }

        System.out.println ("2. rooms=" + rooms + ",   " +
        t1 + ", " + t1.getName() + ", " + t1.getSeats());

        rooms = 123;              //roomsOpen is not changed
        t1.setSeats (17);         //now Java obj has 17 seats

        System.out.println ("3. rooms=" + rooms + ", " +
        t1 + ", " + t1.getName() + ", " + t1.getSeats());

        t1 = new TCourse6 ("UNIX", 9); //t1 points to new obj
        t1.setSeats (14);              //new obj has 14 seats

        System.out.println ("4. rooms=" + rooms + ", " +
        t1 + ", " + t1.getName() + ", " + t1.getSeats());

        return true;
    }
}
```

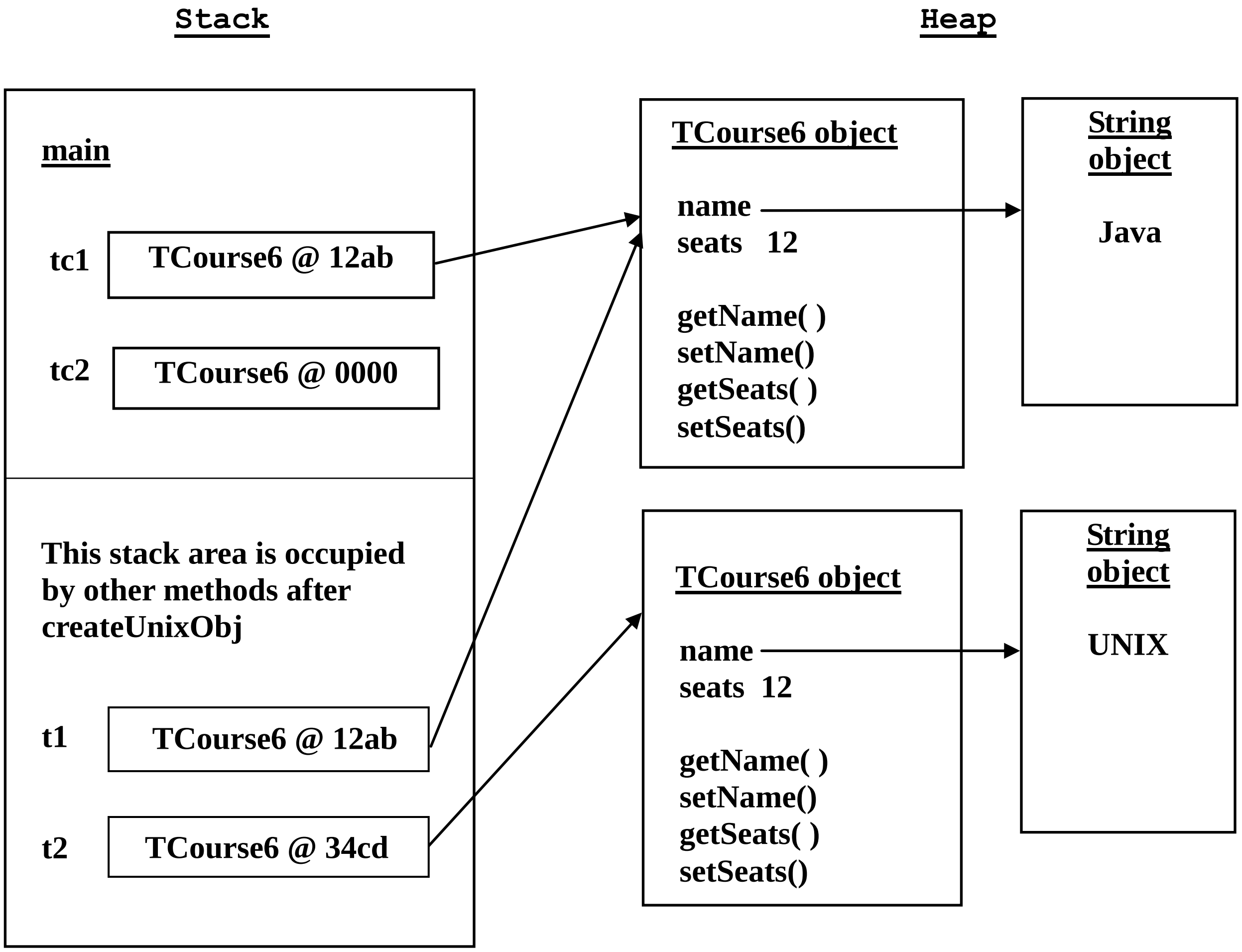
Result, P611.java

```
1. roomsOpen=6, TCourse6@b10190, Java, 12
2. rooms=6,   TCourse6@b10190, Java, 12
3. rooms=123, TCourse6@b10190, Java, 17
4. rooms=123, TCourse6@ddc581, UNIX, 14
5. roomsOpen=6, TCourse6@b10190, Java, 17
```

RETURN A REFERENCE FROM A CALLED METHOD

1. A method can return one item or void.

2. When a method returns an object, in fact a reference to the object is returned. The method header must specify the class type of the object as the return type.



RETURN A REFERENCE FROM A CALLED METHOD, EXAMPLE

P613.java
```
1   public class P613 {
2       public static void main (String[] args) {
3
4           TCourse6 tc1 = new TCourse6 ("Java", 12);
5           TCourse6 tc2 = null;
6
7           tc2 = createUnixObj (tc1);
8
9           if (tc1 != null && tc2 != null) {
10              System.out.println (
11              tc1.getName() + " has " + tc1.getSeats() + ", " +
12              tc2.getName() + " has " + tc2.getSeats() );
13          }
14      }
15
16      public static TCourse6 createUnixObj (TCourse6 t1) {
17          if (t1==null) {
18              return null;
19          }
20
21          TCourse6 t2 = new TCourse6 ("UNIX", 0);     //new obj
22
23          int seatsToBeCopied = t1.getSeats();
24          t2.setSeats (seatsToBeCopied);
25
26          //t2.setSeats (t1.getSeats());  //same as lines 23-24
27
28          return t2;
29      }
30  }
```

Result, P613.java
```
Java has 12, UNIX has 12
```

EXERCISES

1. Create a program called E61.java that makes use of two classes, called E61 and MyString. E61 is the main class.

   In MyString.java

   a. Create a String instance variable called str that is initialized by the constructor to the parameter value.

   b. Create methods getStr and setStr for the variable str.

   In E61.java

   c. Create two objects of type MyString. Display the value in each object. Then, change the value contained in each object, and display them again.

2. Copy E52.java and call the copy E62.java. Then separate E62.java into a main class and a business class as follows:

   a. E62.java will be the main class. It should contain:

```
public class E62 {
    public static void main (String[] args) {

        RoomReservation62 rr1 = new RoomReservation62();
        rr1.setReservationNumber (130323);
        rr1.setSeats (12);
        rr1.setNumberOfDays (5);
        rr1.setDayRatePerSeat (25.00);
        rr1.setTaxRate (0.0725);
        rr1.printOneReservation ();
    }
}
```

   b. Create a business class called RoomReservation62.java.

   c. The private instance variables in RoomReservation62.java include:

      1) variables set by method calls from the main class: reservationNumber, seats, numberOfDays, dayRatePerSeat, and taxRate.

      2) variables to hold calculated amounts: roomAmount, taxAmount, and finalAmount

   d. The constructor should receive no parameters and contain no statements. It should be public.

e. Create a private void method called calculateAmounts that receives no parameters. The method should not be static. This method calculates all amounts.

f. The method printOneReservation should be in this class. The method receives no parameters, calls the method calculateAmounts, and prints the same report as previous versions of this program except add \n newline at the end of the printout. The method should be public and not static.

g. This class should have a pair of get and set methods that are public and not static for each instance variable that is set by a method call from main:

1) getReservationNumber and setReservationNumber
2) getSeats and setSeats
3) getNumberOfDays and setNumberOfDays
4) getDayRatePerSeat and setDayRatePerSeat
5) getTaxRate and setTaxRate

Each get method receives no parameters and returns the value of the specified variable.

Each set method receives one parameter with a value for the specified variable, and returns void. Each set method contains the validation and default value that was coded in the previous version of the program, and assigns the parameter or default value to the specified variable. The method setReservationNumber will not contain any validation.

h. In your main class, create additional RoomReservation62 objects and assign invalid values to the instance variables to test your validation code. For example:

```

    RoomReservation62 rr2 = new RoomReservation62();
    rr2.setReservationNumber (130444);
    rr2.setSeats (15);
    rr2.setNumberOfDays (6);
    rr2.setDayRatePerSeat (66.00);
    rr2.setTaxRate (0.21);
    rr2.printOneReservation ();

    RoomReservation62 rr3 = new RoomReservation62();
    rr3.setReservationNumber (130505);
    rr3.setSeats (7);
    rr3.setNumberOfDays (0);
    rr3.setDayRatePerSeat (24.00);
    rr3.setTaxRate (0.0499);
    rr3.printOneReservation ();
```

SOLUTIONS

E61.java

```
public class E61 {
    public static void main (String[] args) {

        MyString my1 = new MyString ("first");
        MyString my2 = new MyString ("second");

        System.out.println ("before: " +
            my1.getStr() + ", " + my2.getStr() );

        my1.setStr ("new first");
        my2.setStr ("new second");

        System.out.println ("after:  " +
            my1.getStr() + ", " + my2.getStr() );
    }
}
```

MyString.java

```
public class MyString {
    private String str;

    public MyString (String paramString) {
        setStr (paramString);
    }

    public String getStr() {
        return str;
    }
    public void setStr (String s) {
        str = s;
    }
}
```

Result, E61.java

```
before: first, second
after:  new first, new second
```

E62.java

```
public class E62 {
    public static void main (String[] args) {

        RoomReservation62 rr1 = new RoomReservation62();
        rr1.setReservationNumber (130323);
        rr1.setSeats (12);
        rr1.setNumberOfDays (5);
        rr1.setDayRatePerSeat (25.00);
        rr1.setTaxRate (0.0725);
        rr1.printOneReservation ();

        RoomReservation62 rr2 = new RoomReservation62();
        rr2.setReservationNumber (130444);
        rr2.setSeats (15);
        rr2.setNumberOfDays (6);
        rr2.setDayRatePerSeat (66.00);
        rr2.setTaxRate (0.21);
        rr2.printOneReservation ();

        RoomReservation62 rr3 = new RoomReservation62();
        rr3.setReservationNumber (130505);
        rr3.setSeats (7);
        rr3.setNumberOfDays (0);
        rr3.setDayRatePerSeat (24.00);
        rr3.setTaxRate (0.0499);
        rr3.printOneReservation ();
    }
}
```

RoomReservation62.java

```
public class RoomReservation62 {

    private int reservationNumber;
    private int seats;
    private int numberOfDays;
    private double dayRatePerSeat;
    private double taxRate;

    private double roomAmount;
    private double taxAmount;
    private double finalAmount;

    public RoomReservation62 () {
    }

    private void calculateAmounts () {
        roomAmount = seats * numberOfDays * dayRatePerSeat;
        taxAmount = roomAmount * taxRate;
        finalAmount = roomAmount + taxAmount;
    }

    public void printOneReservation () {
        calculateAmounts ();
        System.out.println (
           "Reservation: " + reservationNumber +
         "\nNumber of seats: " + seats +
         "\nNumber of days: " + numberOfDays +
         "\nRoom amount before tax: " + roomAmount +
         "\nTax at " + taxRate*100 + "%: " + taxAmount +
         "\nFinal total: " + finalAmount + "\n");
    }

    public int getReservationNumber () {
        return reservationNumber;
    }
    public void setReservationNumber (int r) {
        reservationNumber = r;
    }

    public int getSeats () {
        return seats;
    }
    public void setSeats (int s) {
        switch (s) {
            case 8:  break;
            case 10: break;
            case 12: break;
            case 14: break;
            default: System.err.println ("Invalid seats "
                         + s + ", will be set to 12");
                     s = 12;
        }
        seats = s;
    }

    public int getNumberOfDays () {
        return numberOfDays;
    }
    public void setNumberOfDays (int n) {
        if (n < 1 || n > 5) {
            System.err.println ("Invalid numberOfDays "
                + n + ", will be set to 5");
            n = 5;
        }
        numberOfDays = n;
    }

    public double getDayRatePerSeat () {
        return dayRatePerSeat;
    }
    public void setDayRatePerSeat (double d){
        if (d < 25.00 || d > 65.00) {
            System.err.println ("Invalid dayRatePerSeat "
                + d + ", will be set to 25.00");
            d = 25.00;
        }
        dayRatePerSeat = d;
    }

    public double getTaxRate () {
        return taxRate;
    }
    public void setTaxRate (double t) {
        if (t < 0.05 || t > 0.20) {
            System.err.println ("Invalid taxRate "
                + t + ", will be set to 0.0725");
            t = 0.0725;
        }
        taxRate = t;
    }
}
```

Result, E62.java

```
Reservation: 130323
Number of seats: 12
Number of days: 5
Room amount before tax: 1500.0
Tax at 7.249999999999999%: 108.74999999999999
Final total: 1608.75

Invalid seats 15, will be set to 12
Invalid numberOfDays 6, will be set to 5
Invalid dayRatePerSeat 66.0, will be set to 25.00
Invalid taxRate 0.21, will be set to 0.0725
Reservation: 130444
Number of seats: 12
Number of days: 5
Room amount before tax: 1500.0
Tax at 7.249999999999999%: 108.74999999999999
Final total: 1608.75

Invalid seats 7, will be set to 12
Invalid numberOfDays 0, will be set to 5
Invalid dayRatePerSeat 24.0, will be set to 25.00
Invalid taxRate 0.0499, will be set to 0.0725
Reservation: 130505
Number of seats: 12
Number of days: 5
Room amount before tax: 1500.0
Tax at 7.249999999999999%: 108.74999999999999
Final total: 1608.75
```

ECLIPSE DEBUGGER

1. Enter the two classes below. MAKE SURE THAT YOUR LINE NUMBERS ARE THE SAME. Execute CustomerTest.java to make sure the code works.

Customer.java

```
1   public class Customer {
2
3       private String name;
4       private double creditLimit;
5
6       public Customer() {
7           super();
8       }
9       public Customer(String name, double creditLimit) {
10          super();
11          setName (name);
12          setCreditLimit (creditLimit);
13      }
14
15      public String getName() {
16          return name;
17      }
18      public void setName(String name) {
19          this.name = name;
20      }
21
22      public double getCreditLimit() {
23          return creditLimit;
24      }
25      public void setCreditLimit(double creditLimit) {
26          this.creditLimit = creditLimit;
27      }
28  }
```

CustomerTest.java

```
1   public class CustomerTest {
2
3       public static void main(String[] args) {
4
5           Customer c = null;
6           c = new Customer ();
7
8           c.setName("Teresa");
9           c.setCreditLimit(100.00);
10
11          String name = c.getName();
12          double creditLimit = c.getCreditLimit();
13
14          System.out.println(
15              "name=" + name +
16              ", creditLimit=" + creditLimit);
17      }
18  }
```

2. In CustomerTest on line 5, right-click in the Editor's blue vertical bar on the left. In the popup box click Toggle Breakpoint. A breakpoint makes execution pause during a Debugging run just prior to executing this line. If your code has no breakpoints, a debug run is the same as a normal run.

3. Click the insect icon to start a debugging run.

=

4. In the popup "Confirm Perspective Switch" click Yes. The perspective will change.

5. In the Editor's left bar on line 5, the breakpoint circle has been overlaid by the debugger's Current Instruction Pointer which is an arrow pointing to the right. This indicates the next line to run. Also, the line is highlighted in green.

6. The Debug view on the top left informs you:
    a. You are running a Java Application.
    b. You are in the class CustomerTest on your localhost.
    c. You are in the main thread which is suspended at the breakpoint on line 5 in CustomerTest.
    d. In the Call Stack you are in the CustomerTest's main method on line 5.
    e. Your JVM is javaw.exe.

7. Click on the Call Stack line for CustomerTest.main line 5.

8. Now the Variables view on the top right informs you:
    a. The local variable args has been created (the gray circle containing L means args is a local variable), and args points to a String[0] array with an Eclipse-generated id.
    b. Your reference c has not been created yet because line 5 has not been executed yet.

9. Locate the step icons (yellow arrows) above the Debug view. They are "Step Into", "Step Over", and "Step Return".

    a. "Step Into" and "Step Over" make execution go ahead one line. If the next code line to be executed calls a method, it will be executed but:

        1) "Step Into" means the debugger will display line-by-line execution of code in the method. You may not be able to step into a method from the API. (Eclipse may have the bytecode, but not the source code).

        2) "Step Over" means the debugger will not display execution of code in the method.

    b. "Step Return" makes the step-by-step debug display jump ahead to the return of the method you are in. The lines in the method are executed but not shown by the debugger.

10. Click the Step Over icon to execute line 5. Now the Editor's line 6 has the Current Instruction Pointer and the green highlight. The Variables view shows that the reference c is local and the Value column shows that c contains null.

11. Click Step Over again to cause line 6 to be executed. Now the Variables view shows that your reference c points to a Customer object with its own Eclipse id. When you click the Expand Button next to c, the variables inside the pointed-to object are listed alphabetically with their current values.

12. The Current Instruction Pointer now points to line 8, which calls the method setName in the Customer object. To show two ways to use the debugger with called methods, we will use Step Over with setName, and Step Into with setCreditLimit.

13. Click Step Over. The Variables view will show the resulting changed value in the <u>name</u> variable in the Customer object.

14. When the Debug Current Instruction Pointer points to c.setCreditLimit(100.00), click Step Into.

15. The Debug view changes so the Call Stack shows that you are inside the Customer.setCreditLimit(double) method on line 26. It was called by CustomerTest.main(String[]) on line 9.

16. The Editor changes to show you the code inside the setCreditLimit method.

17. The Variables view changes to show you the method's local variables.

    a. The <u>this</u> variable points to its own object. By clicking the Expand Button next to <u>this</u> you can see the instance variables of the object.

    b. The assignment on line 26 has not yet been executed to assign the parameter to the instance variable. The setCreditLimit method's local variable, the parameter creditLimit, contains 100.0 but the instance variable creditLimit still contains 0.0.

    c. Click the Expand Button next to the instance variable <u>name</u> to see its hash code. Click the Expand Button next to value to see the elements of the character array that constitutes the String.

    d. Click Step Over (or Step Into). Line 26 executes and assigns the parameter creditLimit to the instance variable creditLimit. Yellow highlight on the instance variable shows where the change occurred. Now both instance variables name and creditLimit are set.

e. Click Step Over or Step Into to leave the setCreditLimit method. The Calls Stack changes to show that you are now back in the CustomerTest class on line 11.

18. When you Step Over twice to execute lines 11 and 12 in CustomerTest.java, the Variables view shows that local variables name and creditLimit in the main method come into existence and contain their appropriate values.

19. When you Step Over several times to reach the main method's closing curly brace and terminate execution of CustomerTest, the Debug view may show the JVM's use of Thread.exit() to exit your application's main thread, and the code of Thread.exit(). Finally the Debug view shows that CustomerTest is terminated.

20. To terminate the debug run prior to the end of the code, in the Debug view click to highlight a line that shows the classname CustomerTest. Right click on the classname to get the popup context menu. Click Terminate. The Debug view should display something like: <terminated, exit value: 0> C:\ ... \javaw.exe

    a. javaw is a version of the JVM java.exe that works as a plug-in. It can run without being in a console window, and displays a popup with messages when it has them.

ADDITIONAL DEBUG TIPS

21. <u>Skip debugging display of the lines in a loop</u>

    a. Create breakpoints just before and just after the loop. When execution stops just before the loop, two ways to make the debugging display skip over the loop:

        1) Click the Resume icon (yellow vertical line and green arrow pointing right) to jump ahead to the next breakpoint which is just after the loop.

        2) Or, put the Editor cursor on the line to jump to, click Run, Run to Line.

22. <u>Modify code in the Editor and save</u> while running in the debugger: With various limitations you can do this.

23. <u>Caution: If you switch to the Java perspective</u> while running the debugger, the debugger will pause but not terminate.

    a, If you have many debugger sessions in paused state, Eclipse will run slowly.

(blank)

UNIT 7: this, NAME COLLISIONS, OVERLOADED CONSTRUCTORS

Upon completion of this unit, students should be able to:

1. Use the keyword this to resolve name conflicts between a method's local variable or parameter, and an instance variable with the same name.

2. Briefly explain the concept of compiletime polymorphism and how it is implemented with the keyword this and overloaded constructors.

3. Code and instantiate classes with overloaded constructors.

this TO RESOLVE NAME COLLISIONS

1. Every object automatically has a variable with the identifier this; the variable is a reference to the current object and has the class type of the current object.

2. The variable this is automatically passed to every instance method when it is called, so the keyword this can be used inside any instance method to qualify the identifier of an instance variable of the same object.

3. Java does not allow two variables to have the same name within the same or nested scopes.

   a. When a method has a local variable (either a parameter or a variable defined within the curlies of the method body) with the same identifier as an instance variable in the class, the method has a "name space collision" which means that the parameter or local variable "hides" the instance variable.

   b. this resolves the collision. this means the identifier refers to the instance variable of the class, not the parameter or local variable of the method.

4. In the set method on the facing page, this is needed because the name of the parameter is the same as the name of the instance variable defined in the class.

5. In the get method on the facing page, this is not needed, but as a matter of style some programmers like to use it.

this TO RESOLVE NAME COLLISIONS, EXAMPLE

P703.java

```
1   public class P703 {
2       public static void main (String[] args) {
3
4           TCourse703 tc = new TCourse703 ("Java");
5
6           tc.setName ("Introduction to Java");
7
8           System.out.println ( tc.getName() );
9       }
10  }
```

TCourse703.java

```
1   public class TCourse703 {
2       private String name;          //instance var, default
3                                     //value is String@000000
4       public TCourse703 (String name) {
5           setName(name);
6       }
7
8       public String getName() {     //no name collision, so
9           return this.name;         //this is not needed.
10      }
11
12      public void setName (String name) {
13          this.name = name;         //parameter has the same
14      }                             //identifier as instance var
15  }                                 //on line 2. this is needed.
```

Result, P703.java

```
Introduction to Java
```

OVERLOADED CONSTRUCTORS, COMPILETIME POLYMORPHISM, DEFAULT VALUES

1. Overloaded methods were covered on pages 5.07-5.08.

2. Overloaded methods have the same name but different parameter lists. Overloaded methods may have the same or different return types.

3. When an overloaded method is called, the Java compiler uses the argument list to select the specific method to be called. Each method, when called, handles its task in its own way.

4. Constructors are typically overloaded. The constructors of the String class are a good example.

5. Default values for instance variables can be created in various ways:

    a. Initialize instance variables in their declarations.

    b. Use a constructor to supply default values if the constructor is called with fewer parameters than are needed to initialize all variables.

    c. Do not explicitly initialize instance variables. In this case, they will contain all bits set to zero.

        1) integer numbers contain 0
        2) floating point numbers contain 0.0
        3) booleans contain false
        4) chars contain '\u0000' (called the null character)
        5) the address portion of reference variables contains null (all bits zero).

OVERLOADED CONSTRUCTORS, COMPILETIME POLYMORPHISM, EXAMPLE

P705.java

```
public class P705 {
    public static void main (String[] args) {

        TCourse705 tc1 = new TCourse705 ();
        TCourse705 tc2 = new TCourse705 ("Java");
        TCourse705 tc3 = new TCourse705 ("UNIX", 10);

        System.out.println ("1. " + tc1.toString() + "  "
          + tc2.toString() + "  " + tc3.toString() );

        tc1.setName ("HTML");
        tc1.setSeats (12);
        tc2.setSeats (14);

        System.out.println ("2. " + tc1.toString() + "  "
          + tc2.toString() + "  " + tc3.toString() );
    }
}
```

TCourse705.java

```
public class TCourse705 {
    private String name = "none";           //default values
    private int seats = -1;                 //via assignment

    public TCourse705 () {
    }
    public TCourse705 (String newName) {
        setName(newName);                    //repeated code
    }
    public TCourse705 (String newName, int newSeats) {
        setName(newName);                    //repeated code
        setSeats(newSeats);
    }

    public String toString () {
        return "TCourse705:" + name + "," + seats;
    }
    public void setName (String name) {
        this.name = name;
    }
    public void setSeats (int seats) {
        this.seats = seats;
    }
}
```

Result, P705.java

```
1. TCourse705:none,-1  TCourse705:Java,-1  TCourse705:UNIX,10
2. TCourse705:HTML,12  TCourse705:Java,14  TCourse705:UNIX,10
```

OVERLOADED CONSTRUCTORS WITH this, COPY AN OBJECT

1. To avoid duplication of code, one overloaded method can call another within the same class by specifying the method name and passing the desired arguments.

2. For an overloaded constructor to call another constructor in the same class, you must use the keyword this to serve as the constructor name. The call to this must be the first statement in the constructor.

3. The keyword this is a reference to the current object. It can be used in any method or constructor, and has two primary uses:

    a. As a constructor name, this enables one overloaded constructor to call another during construction of an object.

    b. If a constructor or method has local variables (variables defined within the curlies of the method body, or parameters) with the same name as instance variables of its class, this can be used as a qualifier to refer to the instance variables of the class.

4. To create a copy of an object, you must create a new object:

```
TCourse707 tc4 = new TCourse707 (tc2);
```

OPTIONAL NOTES

5. In a static method, it is a compile error to code a reference to a static variable as this.varName.

6. Static methods are not allowed to access instance variables or methods without using a qualifier that consists of a reference to an existing object, because static methods may be called when no instance of the class has been instantiated.

7. In an instance method, you can refer to a static variable as this.varName, but it is proper style to use the classname as qualifier.

OVERLOADED CONSTRUCTORS WITH this, COPY AN OBJECT, EXAMPLE

P707.java

```
public class P707 {
    public static void main (String[] args) {
        TC707 tc1 = new TC707 ("UNIX", 10);
        TC707 tc2 = new TC707 ("Java");
        TC707 tc3 = new TC707 ();
        TC707 tc4 = new TC707 (tc2);
        //tc4 = new TC707 (tc2.getName(), tc2.getSeats());

        System.out.println (
            tc1.toString() + " " + tc2.toString() + " " +
            tc3.toString() + " " + tc4.toString() );
    }
}
```

TC707.java

```
public class TC707 {
    private String name;   //no initial values are coded, so
    private int seats;     //vars will have all zero bits
                           //until code assigns other values
    public TC707 (String name, int seats) {
        this.name = name;
        this.seats = seats;
    }
    public TC707 (String name) {
        this (name, -1);         //same as  this.name=name;
    }                            //         this.seats=-1;
    public TC707 () {
        this ("none");
    }
    public TC707 (TC707 tc) {    //copy constructor
        this (tc.getName(), tc.getSeats());
    }

    public String toString () {
        return "TC707:" + name + "," + seats;
    }
    public String getName () {
        return name;
    }
    public int getSeats () {
        return seats;
    }
}
```

Result, P707.java

```
TC707:UNIX,10 TC707:Java,-1 TC707:none,-1 TC707:Java,-1
```

EXERCISES

1. Modify your program E61.java and call the modified version E71.java.

   a. Rename class MyString to MyString7. In the set method, give the parameter the same identifier as the instance variable, and use this to resolve the name collision.

   b. Create an overloaded MyString7 constructor that receives no parameters and calls the other constructor with a default of "default string".

   c. In your main method rename MyString to MyString7. Create a MyString7 object without passing an argument to the constructor.

2. Create a new class E72.java. Copy RoomReservation62.java, modify it, and call the new version RoomReservation72.java.

   E72.java should contain the following.

```
public class E72 {
    public static void main (String[] args) {

        RoomReservation72 rr323 = new RoomReservation72 (
            130323, 12, 5, 25.00, 0.0725);
        rr323.printOneReservation ();

        RoomReservation72 rr444 = new RoomReservation72 (
            130444, 14, 3, 35.00);           //no taxRate
        rr444.printOneReservation ();
    }
}
```

   In RoomReservation72.java

   a. Create a public constructor that receives five parameters and calls the appropriate set methods to initialize the five instance variables that are not calculated amounts.

   b. Create a public constructor that receives four parameters and passes them, along with the double literal 0.0725 as the default taxRate, to the constructor described in a. (Now a constructor and the method setTaxRate contain the default tax rate 0.0725, which is duplicate code that will be eliminated in the next unit.)

   c. Modify your set methods so the same name is used for the parameter received and the instance variable to be set. Use this to resolve the name collisions.

ECLIPSE

1. Getters and setters

    a. After your instance variable declarations have been entered, place the cursor on the line before where you want the getters amd setters. Click Source, Generate Getters and Setters...

    b. Click checkboxes for variables that need get and set methods. (If you click the expand button (square with +) in front of the variable names, the method names are displayed. These names comply with the JavaBeans standard, which is the industry standard.

    c. Click OK.

2. Constructors

    a. Place your cursor on the line before where you want a constructor. Click Source, Generate Constructor using Fields...

    b. Click checkboxes for variables to be initialized. For a null constructor Deselect All. Click OK.

    c. Current versions of Eclipse include the variables in the order they are declared.

    d. You must manually modify the assignments to call your set methods.

SOLUTIONS

E71.java

```java
public class E71 {
    public static void main (String[] args) {

        MyString7 my1 = new MyString7 ("first");
        MyString7 my2 = new MyString7 ();

        System.out.println ("before: " +
            my1.getStr() + ", " + my2.getStr() );

        my1.setStr ("new first");
        my2.setStr ("new second");

        System.out.println ("after:  " +
            my1.getStr() + ", " + my2.getStr() );
    }
}
```

MyString7.java

```java
public class MyString7 {
    private String str;

    public MyString7 (String paramString) {
        setStr (paramString);
    }
    public MyString7 () {
        this ("default string");
    }

    public String getStr() {
        return str;
    }
    public void setStr (String str) {
        this.str = str;
    }
}
```

Result, E71.java

```
before: first, default string
after:  new first, new second
```

E72.java

```
public class E72 {
    public static void main (String[] args) {

        RoomReservation72 rr323 = new RoomReservation72 (
            130323, 12, 5, 25.00, 0.0725);
        rr323.printOneReservation ();

        RoomReservation72 rr444 = new RoomReservation72 (
            130444, 14, 3, 35.00);           //no taxRate
        rr444.printOneReservation ();
    }
}
```

RoomReservation72.java

```
public class RoomReservation72 {

    private int reservationNumber;
    private int seats;
    private int numberOfDays;
    private double dayRatePerSeat;
    private double taxRate;

    private double roomAmount;
    private double taxAmount;
    private double finalAmount;


    public RoomReservation72 () {
    }
    public RoomReservation72 (
        int reservationNumber, int seats, int numberOfDays,
        double dayRatePerSeat, double taxRate
        ) {
        setReservationNumber (reservationNumber);
        setSeats (seats);
        setNumberOfDays (numberOfDays);
        setDayRatePerSeat (dayRatePerSeat);
        setTaxRate (taxRate);
    }
    public RoomReservation72 (
        int reservationNumber, int seats, int numberOfDays,
        double dayRatePerSeat
        ) {
        this (reservationNumber, seats, numberOfDays,
            dayRatePerSeat, 0.0725);
    }


    private void calculateAmounts () {
        roomAmount = seats * numberOfDays * dayRatePerSeat;
        taxAmount = roomAmount * taxRate;
        finalAmount=roomAmount + taxAmount;
    }
    public void printOneReservation () {
        calculateAmounts ();
        System.out.println (
           "Reservation: " + reservationNumber +
         "\nNumber of seats: " + seats +
         "\nNumber of days: " + numberOfDays +
         "\nRoom amount before tax: " + roomAmount +
         "\nTax at " + taxRate*100 + "%: " + taxAmount +
         "\nFinal total: " + finalAmount + "\n");
    }


    public int getReservationNumber () {
        return reservationNumber;
    }
    public void setReservationNumber(int reservationNumber) {
        this.reservationNumber = reservationNumber;
    }


    public int getSeats () {
        return seats;
    }
    public void setSeats (int seats) {
        switch (seats) {
            case 8:  break;
            case 10: break;
            case 12: break;
            case 14: break;
            default: System.err.println ("Invalid seats "
                         + seats + ", will be set to 12");
                     seats = 12;
        }
        this.seats = seats;
    }


    public int getNumberOfDays () {
        return numberOfDays;
    }
    public void setNumberOfDays (int numberOfDays) {
        if (numberOfDays < 1 || numberOfDays > 5) {
            System.err.println ("Invalid numberOfDays "
                + numberOfDays + ", will be set to 5");
            numberOfDays = 5;
        }
        this.numberOfDays = numberOfDays;
    }


    public double getDayRatePerSeat() {
        return dayRatePerSeat;
    }
    public void setDayRatePerSeat(double dayRatePerSeat) {
        if (dayRatePerSeat<25.00 || dayRatePerSeat>65.00) {
            System.err.println ("Invalid dayRatePerSeat "
                + dayRatePerSeat + ", will be set to 25.00");
            dayRatePerSeat = 25.00;
        }
        this.dayRatePerSeat = dayRatePerSeat;
    }


    public double getTaxRate() {
        return taxRate;
    }
    public void setTaxRate(double taxRate) {
        if (taxRate < 0.05 || taxRate > 0.20) {
            System.err.println ("Invalid taxRate "
                + taxRate + ", will be set to 0.0725");
            taxRate = 0.0725;
        }
        this.taxRate = taxRate;
    }
}
```

Result, E72.java

```
Reservation: 130323
Number of seats: 12
Number of days: 5
Room amount before tax: 1500.0
Tax at 7.249999999999999%: 108.74999999999999
Final total: 1608.75

Reservation: 130444
Number of seats: 14
Number of days: 3
Room amount before tax: 1470.0
Tax at 7.249999999999999%: 106.57499999999999
Final total: 1576.575
```

E72Alternate.java

```
//Alternate style main class to test ctors and methods:
public class E72Alternate {
    public static void main (String[] args) {

        p ("\nTest constructor with good data");
        p (  "-------------------------------");
        RoomReservation72 rr323 = new RoomReservation72 (
            130323, 12, 5, 25.00, 0.0725);

        p ("setReservationNumber, expected: 130323");
        p ("getReservationNumber, actual:   " +
            rr323.getReservationNumber() );
        p ("setSeats, expected: 12");
        p ("getSeats, actual:   " + rr323.getSeats() );
        p ("setNumberOfDays, expected: 5");
        p ("getNumberOfDays, actual:   " +
            rr323.getNumberOfDays() );
        p ("setDayRatePerSeat, expected: 25.00");
        p ("getDayRatePerSeat, actual:   " +
            rr323.getDayRatePerSeat() );
        p ("setTaxRate, expected: 0.0725");
        p ("getTaxRate, actual:   " +
            rr323.getTaxRate() );


        p ("\nTest default tax rate");
        p (  "---------------------");
        RoomReservation72 rr444 = new RoomReservation72 (
            130444, 14, 3, 35.00);           //no taxRate

        p ("Test default tax rate, expected: 0.0725");
        p ("getTaxRate, actual:              " +
            rr444.getTaxRate() );


        p ("\nTest individual methods with good data");
        p (  "--------------------------------------");
        RoomReservation72 rr505 = new RoomReservation72 ();

        int reservationNumber = 130505;
        int seats = 14;
        int numberOfDays = 3;
        double dayRatePerSeat = 45.00;
        double taxRate = 0.0650;

        rr505.setReservationNumber (reservationNumber);
        rr505.setSeats(seats);
        rr505.setNumberOfDays(numberOfDays);
        rr505.setDayRatePerSeat(dayRatePerSeat);
        rr505.setTaxRate(taxRate);

        p ("setReservationNumber, expected: " +
            reservationNumber);
        p ("getReservationNumber, actual:   " +
            rr505.getReservationNumber() );
        p ("setSeats, expected: " + seats);
        p ("getSeats, actual:   " + rr505.getSeats() );
        p ("setNumberOfDays, expected: " + numberOfDays);
        p ("getNumberOfDays, actual:   " +
            rr505.getNumberOfDays() );
        p ("setDayRatePerSeat, expected: " + dayRatePerSeat);
        p ("getDayRatePerSeat, actual:   " +
            rr505.getDayRatePerSeat() );
        p ("setTaxRate, expected: " + taxRate);
        p ("gettaxRate, actual:   " +
            rr505.getTaxRate() );
    }
    public static void p (String s) {
        System.out.println (s);
    }
}
```

Result, E72Alternate.java

```
Test constructor with good data
-------------------------------
setReservationNumber, expected: 130323
getReservationNumber, actual:   130323
setSeats, expected: 12
getSeats, actual:   12
setNumberOfDays, expected: 5
getNumberOfDays, actual:   5
setDayRatePerSeat, expected: 25.00
getDayRatePerSeat, actual:   25.0
setTaxRate, expected: 0.0725
getTaxRate, actual:   0.0725

Test default tax rate
--------------------
Test default tax rate, expected: 0.0725
getTaxRate, actual:              0.0725

Test individual methods with good data
--------------------------------------
setReservationNumber, expected: 130505
getReservationNumber, actual:   130505
setSeats, expected: 14
getSeats, actual:   14
setNumberOfDays, expected: 3
getNumberOfDays, actual:   3
setDayRatePerSeat, expected: 45.0
getDayRatePerSeat, actual:   45.0
setTaxRate, expected: 0.065
gettaxRate, actual:   0.065
```

(blank)

UNIT 8: STATIC AND INSTANCE MEMBERS, PUBLIC AND PRIVATE

Upon completion of this unit, students should be able to:

1. Code and instantiate classes with both static and instance variables and methods.

2. Briefly explain the modifiers public and private, and create classes that use them.

MODIFIER static, STATIC AND INSTANCE MEMBERS

1. The variables and methods in a class (called the members of the class) may be instance or static. Static members are also called class members.

    a. Instance members are instantiated in each object of the class. Each object contains a complete set of all instance members.

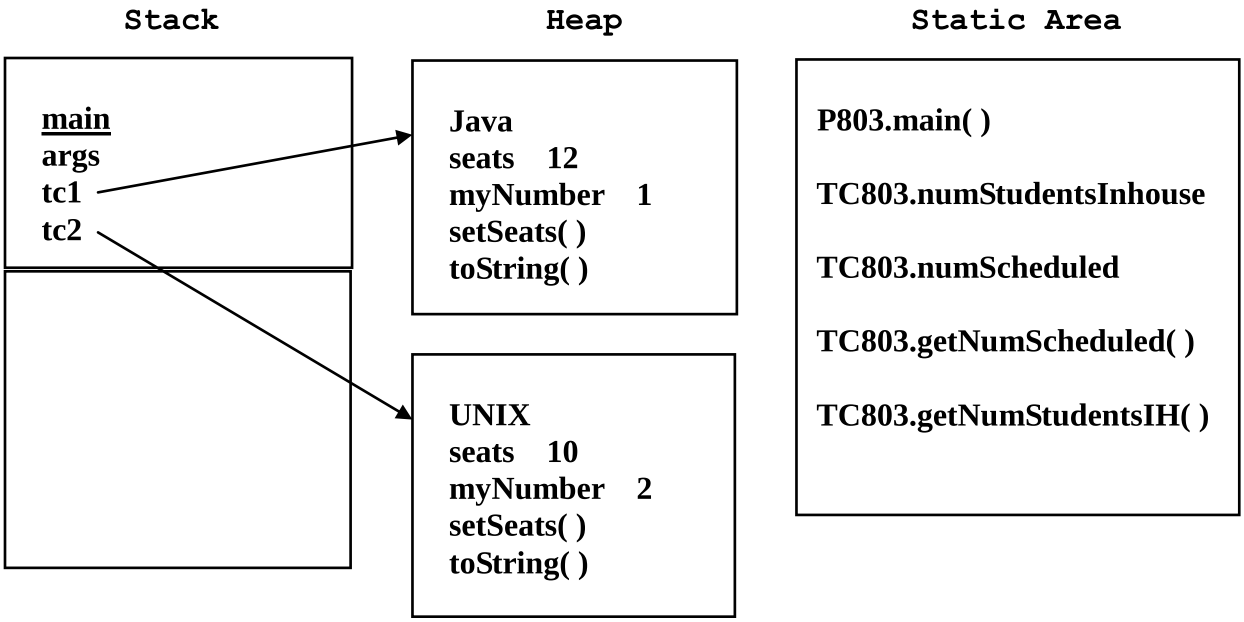
    b. Static members are not associated with any object. During runtime, when the name of a class is mentioned in the program for the first time, the JVM loads the class, scans it for static members, and creates the static members in a separate part of storage called the static area which is in or near the heap.

        1) Static members are coded with the keyword static.

        2) In a program, all methods of all objects can access all accessible static items.

        3) Static variables and methods are qualified by their classname. One example is the method System.exit().

2. Static members can be used even though no object of their class exists. This is why the main method is static. When you give the JVM the name of your main class to be executed, its static members are created in the static area. This makes the main method available to be called, and program execution can begin.



MODIFIER static, STATIC AND INSTANCE MEMBERS, EXAMPLE

P803.java

```java
public class P803 {
    public static void main (String[] args) {

        System.out.println ("Number of courses scheduled: " +
            TC803.getNumScheduled() );

        TC803 tc1 = new TC803 ("Java", 12);
        TC803 tc2 = new TC803 ("UNIX", 10);

        System.out.println ("Number of courses today: " +
            TC803.getNumScheduled() + ":  " +
            tc1.toString() + "  " + tc2.toString() + "\n" +
            "Number of students in training center: " +
            TC803.getNumStudentsInHouse() );
    }
}
```

TC803.java

```java
public class TC803 {
    private static int numStudentsInHouse;//total accumulator
    private static int numScheduled;      //number generator
    private int myNumber;
    private String name;
    private int seats;

    public TC803 (String name, int seats) {
        this.name = name;
        setSeats (seats);
        numScheduled++;
        myNumber = numScheduled;
    }
    public static int getNumScheduled() {
        return numScheduled;
    }
    public static int getNumStudentsInHouse() {
        return numStudentsInHouse;
    }
    public void setSeats (int seats) {
        this.seats = seats;
        numStudentsInHouse = numStudentsInHouse + seats;
    }
    public String toString () {
       return "TC803:" + myNumber + "," + name + "," + seats;
    }
}
```

Result, P803.java

```
Number of courses scheduled: 0
Number of courses today: 2:  TC803:1,Java,12  TC803:2,UNIX,10
Number of students in training center: 22
```

ACCESS CONTROL VIA public AND private

1. Access control means that each class controls whether or not other classes in the same program can directly perform these actions:

   a. Obtain or modify the values of variables.

   b. Call methods.

2. Access control is done via the modifiers public, protected, and private.

   a. public means that your variable or method may be accessed by code in any other class in the same program.

   b. protected means that your variable or method may be accessed by code in another class in the same program IF that class is in same package as your class, or is a subclass of your class. Packages and subclasses are covered in later units.

   c. private means that your variable or method may be accessed only by other members of the same class.

3. A variable or method with no access modifier has default access, also called "package friendly" access because the variable or method may be accessed by code in classs in the same package. Packages are covered in a later unit.

4. The purpose of access control is to prevent errors in the use of variables or methods by allowing access to them only via public or protected members of other classes in the same program.

5. Encapsulation of private members, and creation of a class's public interface, are implemented via access control.

EXERCISES

1. Create a main class called E82.java that instantiates three RoomReservation82 objects so that each constructor of RoomReservation82.java is called one time. For two objects, pass invalid data values to the constructor.

   In this suggested solution, lines 4-18 were copied from E62.java and then modified.

```
1   public class E82 {
2       public static void main (String[] args) {
3
4           RoomReservation82 rr1 = new RoomReservation82();
5           rr1.setReservationNumber (130323);
6           rr1.setSeats (12);
7           rr1.setNumberOfDays (5);
8           rr1.setDayRatePerSeat (25.00);
9           rr1.setTaxRate (0.0725);
10          rr1.printOneReservation ();
11
12          RoomReservation82 rr2 = new RoomReservation82 (
13              130444, 15, 6, 66.00, 0.21);  //invalid data
14          rr2.printOneReservation ();
15
16          RoomReservation82 rr3 = new RoomReservation82 (
17              130505, 7, 0, 24.00);         //invalid data
18          rr3.printOneReservation ();
19      }
20  }
```

2. Copy RoomReservation72.java and call the copy RoomReservation82.java. In RoomReservation82.java replace the hard-coded literal default values with constants as shown below. Make these changes one at a time, and run the program to confirm that it works before going on to the next change.

```
public static final int    DEFAULT_SEATS = 12;
public static final int    DEFAULT_NUMBER_OF_DAYS = 5;
public static final double DEFAULT_DAY_RATE_PER_SEAT = 25.00;
public static final double DEFAULT_TAX_RATE = 0.0725;
```

SOLUTIONS

RoomReservation82.java

```
public class RoomReservation82 {

    public static final int    DEFAULT_SEATS = 12;
    public static final int    DEFAULT_NUMBER_OF_DAYS = 5;
    public static final double DEFAULT_DAY_RATE_PER_SEAT
        = 25.00;
    public static final double DEFAULT_TAX_RATE = 0.0725;

    private int reservationNumber;
    private int seats;
    private int numberOfDays;
    private double dayRatePerSeat;
    private double taxRate;

    private double roomAmount;
    private double taxAmount;
    private double finalAmount;


    public RoomReservation82 () {
    }
    public RoomReservation82 (
        int reservationNumber, int seats, int numberOfDays,
        double dayRatePerSeat, double taxRate
        ) {
        setReservationNumber (reservationNumber);
        setSeats (seats);
        setNumberOfDays (numberOfDays);
        setDayRatePerSeat (dayRatePerSeat);
        setTaxRate (taxRate);
    }
    public RoomReservation82 (
        int reservationNumber, int seats, int numberOfDays,
        double dayRatePerSeat
        ) {
        this (reservationNumber, seats, numberOfDays,
            dayRatePerSeat, DEFAULT_TAX_RATE);
    }


    private void calculateAmounts () {
        roomAmount = seats * numberOfDays * dayRatePerSeat;
        taxAmount = roomAmount * taxRate;
        finalAmount=roomAmount + taxAmount;
    }

    public void printOneReservation () {
        calculateAmounts ();
        System.out.println (
           "Reservation: " + reservationNumber +
         "\nNumber of seats: " + seats +
         "\nNumber of days: " + numberOfDays +
         "\nRoom amount before tax: " + roomAmount +
         "\nTax at " + taxRate*100 + "%: " + taxAmount +
         "\nFinal total: " + finalAmount + "\n");
    }


    public int getReservationNumber () {
        return reservationNumber;
    }
    public void setReservationNumber(int reservationNumber) {
        this.reservationNumber = reservationNumber;
    }


    public int getSeats () {
        return seats;
    }
    public void setSeats (int seats) {
        switch (seats) {
            case 8:  break;
            case 10: break;
            case 12: break;
            case 14: break;
            default: System.err.println ("Invalid seats "
                         + seats + ", will be set to "
                         + DEFAULT_SEATS);
                     seats = DEFAULT_SEATS;
        }
        this.seats = seats;
    }


    public int getNumberOfDays () {
        return numberOfDays;
    }
    public void setNumberOfDays (int numberOfDays) {
        if (numberOfDays < 1 || numberOfDays > 5) {
            System.err.println ("Invalid numberOfDays "
                + numberOfDays + ", will be set to "
                + DEFAULT_NUMBER_OF_DAYS);
            numberOfDays = DEFAULT_NUMBER_OF_DAYS;
        }
        this.numberOfDays = numberOfDays;
    }


    public double getDayRatePerSeat() {
        return dayRatePerSeat;
     }
     public void setDayRatePerSeat(double dayRatePerSeat) {
        if (dayRatePerSeat<25.00 || dayRatePerSeat>65.00) {
            System.err.println ("Invalid dayRatePerSeat "
                + dayRatePerSeat + ", will be set to "
                + DEFAULT_DAY_RATE_PER_SEAT);
            dayRatePerSeat = DEFAULT_DAY_RATE_PER_SEAT;
        }
        this.dayRatePerSeat = dayRatePerSeat;
    }


    public double getTaxRate() {
        return taxRate;
    }
    public void setTaxRate(double taxRate) {
        if (taxRate < 0.05 || taxRate > 0.20) {
            System.err.println ("Invalid taxRate " + taxRate
                + ", will be set to " + DEFAULT_TAX_RATE);
            taxRate = DEFAULT_TAX_RATE;
        }
        this.taxRate = taxRate;
    }
}
```

Result, E82.java

```
Reservation: 130323
Number of seats: 12
Number of days: 5
Room amount before tax: 1500.0
Tax at 7.249999999999999%: 108.74999999999999
Final total: 1608.75

Invalid seats 15, will be set to 12
Invalid numberOfDays 6, will be set to 5
Invalid dayRatePerSeat 66.0, will be set to 25.0
Invalid taxRate 0.21, will be set to 0.0725
Reservation: 130444
Number of seats: 12
Number of days: 5
Room amount before tax: 1500.0
Tax at 7.249999999999999%: 108.74999999999999
Final total: 1608.75

Invalid seats 7, will be set to 12
Invalid numberOfDays 0, will be set to 5
Invalid dayRatePerSeat 24.0, will be set to 25.0
Reservation: 130505
Number of seats: 12
Number of days: 5
Room amount before tax: 1500.0
Tax at 7.249999999999999%: 108.74999999999999
Final total: 1608.75
```

UNIT 9: INHERITANCE AND ABSTRACT CLASSES

Upon completion of this unit, students should be able to:

1. Briefly explain the concept of inheritance, and how it is implemented in Java.

2. Briefly explain the concepts of overriding and overloading, and the difference between them.

3. Use the keyword super to call the constructor of the immediate superclass.

4. Use the keyword super to refer to accessible hidden variables and overridden methods of the immediate superclass.

5. Code abstract classes and abstract methods, and briefly explain their purpose.

INHERITANCE, CONSTRUCTORS OF SUPERCLASSES

1. Inheritance is a system of organized re-use of the variables and methods in classes.

2. Inheritance terminology:

superclass	subclass
parent	child
ancestor	descendent
base class	derived class

3. A superclass is a more generalized class containing the variables and methods that a group of subclasses have in common. Each subclass inherits the variables and methods of all its ancestors, and can use the identifiers of those that are non-private. A subclass only has to contain the specialized variables and methods that differentiate it from its parent and other subclasses.

4. Constructors are not inherited.

5. In designing classes, there should be a separation of functionality, so that each class has a coherent "identity" and does one thing well. Common functionality of a group of classes should be gathered into a superclass.

6. The Java 7 API is organized into a hierarchy of 4025 classes that descend from the top superclass, Object. All classes derive from Object. (All basic data types are defined in the compiler and JVM, and are not derived from any class.)

7. A subclass can have only one immediate superclass, the name of which is specified via an extends clause in the subclass header. A class defined without an extends clause gets Object in java.lang as its superclass.

8. When a subclass object is created, javac creates a call stack with step by step calls to a constructor in each ancestor all the way up the inheritance tree to Object. During execution, these constructors are executed from Object on down.

9. Each ancestor must have a constructor with parameters that match the arguments passed to it in the call stack. If a constructor does not explicitly call a constructor of its superclass, javac will insert a constructor call that passes no arguments. In this case the program will not compile unless the superclass has a constructor that accepts no parameters. A constructor that accepts no parameters may be called a "null constructor."

10. If a class is coded with no constructor javac gives it a default constructor that accepts no parameters and calls super with no arguments.

extends AND super(), EXAMPLE

P903.java

```java
class I {
    private int i;
    public I (int i) { this.i = i; }
    public int getI () { return i; }
}

class J extends I {
    private int j;
    public J (int i, int j) { super(i); this.j=j; }
    public int getIJ () { return getI() + j; }
}

class K extends J {
    private int k;
    public K (int i, int j, int k) { super(i, j); this.k=k; }
    public int getIJK () { return getIJ() + k; }
}

public class P903 {
    public static void main (String[] args) {
        I refI = new I (1);
        J refJ = new J (10, 20);
        K refK = new K (100, 200, 300);

        System.out.println ("I=" + refI.getI() +
                          ", J=" + refJ.getIJ() +
                          ", K=" + refK.getIJK());
    }
}
```

Result, P903.java

```
I=1, J=30, K=600
```

==

1. To properly initialize superclass variables, private or not, code super() as the first statement in a subclass constructor to call the IMMEDIATE superclass’s constructor. The arguments passed with super() must match one of the superclass's constructors.

2. Subclasses should not repeat the code in their parent class. The parent class should validate, calculate, or initialize its own variables.

3. If inherited variables are private, they are not accessible by name and should be accessed via their get and set methods.

ACCESS TO INHERITED MEMBERS, OVERRIDING VERSUS OVERLOADING

1. A subclass inherits ALL non-static variables and methods from ALL its ancestors including private variables and methods.

2. A subclass can call all the accessible (non-private) methods that it inherits, as if they were coded in the subclass, except:

    a. If the subclass <u>overrides</u> an accessible superclass method by defining its own method with the same name, parameter list and return type, then the subclass must use the keyword <u>super</u> to call the inherited overridden method.

    ```
    super.methodName()
    ```

3. A subclass can use all the accessible (non-private) variables that it inherits, as if they were coded in the subclass, except:

    a. If the subclass <u>hides</u> or <u>shadows</u> an accessible superclass variable by defining its own variable with the same name, then the subclass must use the keyword <u>super</u> to access the inherited hidden variable.

    ```
    super.varName
    ```

    b. It would be very unusual for one class to access the variables of another class (other than public static final variables) except via public get and set methods, because this breaks encapsulation.

4. OVERRIDING VERSUS OVERLOADING METHODS:

    a. A subclass method with the same name, parameter list and return type OVERRIDES an ancestor's method.

    b. A subclass method with the same name but different parameter list OVERLOADS an ancestor's method. Java calls the correct method based on the arguments passed. Overloaded methods may have the same or different return types, but typically have the same.

5. Overriding always involves a subclass and superclass, but overloading can be done within the same class or between a subclass and superclass.

6. Static methods are implicitly final and cannot be overridden. An overriding method cannot narrow the accessibility of the method it overrides, and cannot add new Exceptions.

OVERRIDING VERSUS OVERLOADING, EXAMPLE

P905.java

```
1   class I {
2       private int i;
3       public I (int i) {
4           this.i=i;
5       }
6       public int getTot () {
7           return i;
8       }
9   }
10
11  class J1 extends I { //OVERRIDE--REPLACE I's getTot
12      private int j1;
13      public J1 (int i, int j1) {
14          super(i);                       //super, call line 3
15          this.j1=j1;
16      }
17      public int getTot () {
18          return super.getTot() + j1;     //super, call line 6
19      }                                   //danger--recursion,
20  }                                       //StackOverflowError
21
22  class J2 extends I { //OVERLOAD--AN ALTERNATIVE TO I's getTot
23      private int j2;
24      public J2 (int i, int j2) {
25          super(i);                       //super, call line 3
26          this.j2=j2;
27      }
28      public int getTot (int arg) {
29          return super.getTot() + j2 + arg;//super, call line 6
30      }
31  }
32
33  public class P905 {
34      public static void main (String[] args) {
35          I objI   = new I (1);
36          J1 objJ1 = new J1 (10, 20);
37          J2 objJ2 = new J2 (100, 200);
38
39          System.out.println ("objI. "     + objI.getTot()  );
40          System.out.println ("objJ1. "    + objJ1.getTot() );
41          System.out.println ("objJ2. "    + objJ2.getTot() );
42          System.out.println ("objJ2(5). " + objJ2.getTot(5));
43      }
44  }
```

Result, P905.java

```
objI. 1
objJ1. 30
objJ2. 100
objJ2(5). 305
```

ABSTRACT CLASSES AND METHODS

1. An abstract class is typically used to standardize the method names in subclasses, and to provide the setup for runtime polymorphism (covered in Unit 11).

2. An abstract class can specify abstract method headers for methods that each subclass must implement (override) or the compiler will not compile the subclass.

   a. Often the abstract superclass is not specific or complete enough to be useful by itself, such as a BankAccount class that has Savings and Checking as subclasses.

   b. Concrete (non-abstract) subclasses must contain or inherit concrete implementations for all abstract methods in their abstract superclasses. (A grandchild can inherit concrete implementations from its parent which is a child of the abstract class.)

   c. A subclass that does not have implementations for all the abstract methods in its ancestors must be declared abstract.

3. A class is declared abstract by coding the keyword abstract in its header. A class that contains one or more abstract methods must be declared abstract.

4. An abstract class may (but is not required to) contain abstract methods. An abstract class may (but is not required to) contain non-abstract ("concrete") methods.

5. Abstract method declarations:

   a. Have the keyword abstract in their header.
   b. Do not have a body (the method header must be followed by ; semicolon rather than { } curly braces).
   c. Must be overridden by all concrete (non-abstract) subclasses.
   d. Cannot be private or static, because private and static methods cannot be overridden.

6. An abstract class cannot be instantiated.

7. An abstract superclass is part of the inheritance hierarchy, and its constructor will be called as part of the series of constructors called when a subclass object is created. If you do not code a constructor, Java provides a default one.

8. References of abstract class types are often used to point to objects of its concrete subclasses. We will see this in Unit 11 when we cover runtime polymorphism.

ABSTRACT CLASSES AND METHODS, EXAMPLE

P907.java

```java
public class P907 {
    public static void main (String[] args) {
        ComputerCourse cc = new ComputerCourse ("Java");
        SalesCourse sc = new SalesCourse ("New Clients");
        System.out.println (cc.getNameString() );
        System.out.println (sc.getNameString() );
    }
}

abstract class TCourse9  {
    public TCourse9 () {
    }
    public abstract String getNameString () ;
    public abstract void setNameString (String name) ;
}

class ComputerCourse extends TCourse9 {
    private String name;
    public ComputerCourse (String name) {
        setNameString (name);
    }
    public String getNameString () {
        return "Computer Course is \"" + name + "\"";
    }
    public void setNameString (String name) {
        if ( name.equals("Java") || name.equals("UNIX") )
            this.name = name;
    }
}

class SalesCourse extends TCourse9 {
    private String name;
    public SalesCourse (String name) {
        setNameString (name);
    }
    public String getNameString () {
        return "Sales Course is \"" + name + "\"";
    }
    public void setNameString (String name) {
        if ( name.equals("New Clients")
        ||   name.equals("Referrals") )
            this.name = name;
    }
}
```

Result, P907.java

```
Computer Course is "Java"
Sales Course is "New Clients"
```

EXERCISES

1. Create a program called E91.java with a superclass called MySuper and a subclass called MySub. In addition to the methods described below, create any other methods needed to accomplish these tasks.

   In MySuper

   a. Create an int instance variable that is initialized by a constructor that accepts one int parameter.

   In MySub

   b. Create an int instance variable.

   c. The MySub constructor should be overloaded.

      1) If two int values are passed to it, the first int is used to initialize the int in MySuper and the second int is used to initialize the int in MySub.

      2) If only one int value is passed to it, the int in MySuper is initialized to the parameter, and the int in MySub is initialized to the sum of the parameter plus 1.

   In E91 in main

   d. Create a MySuper object called parent with the int value 91, and display its contents.

   e. Create a MySub object called child1 with the int values 11 and 12, and display its contents.

   f. Create a MySub object called child2 with an int value of 22, and display its contents.

2. Copy E82.java which creates three RoomReservation82 objects, and call the copy E92.java. RoomReservation82.java will be used without changes, but we will create a subclass called FoodService.java to extend it.

   E92.java

   a. Comment out the lines that created objects of class RoomReservation82. We will use these lines again in a later exercise.

b. Create two FoodService objects and, for each one, call its printOneReservation method.

```
public class E92 {
    public static void main (String[] args) {

    //  RoomReservation82 rr1 = new RoomReservation82();
    //  rr1.setReservationNumber (130323);
    //  rr1.setSeats (12);
    //  rr1.setNumberOfDays (5);
    //  rr1.setDayRatePerSeat (25.00);
    //  rr1.setTaxRate (0.0725);
    //  rr1.printOneReservation ();
    //
    //  RoomReservation82 rr2 = new RoomReservation82 (
    //      130444, 15, 6, 66.00, 0.21);  //invalid data
    //  rr2.printOneReservation ();
    //
    //  RoomReservation82 rr3 = new RoomReservation82 (
    //      130505, 7, 0, 24.00);         //invalid data
    //  rr3.printOneReservation ();

        FoodService fs614 = new FoodService (
            130614, 12, 5, 45.00, 0.0725, true, true);
        fs614.printOneReservation ();

        FoodService fs782 = new FoodService (
            130782, 14, 3, 35.00, true, false);
        fs782.printOneReservation ();
    }
}
```

FoodService.java should contain:

c. Constants.

```
public static final double AM_FOOD_COST_PER_PERSON=9.00;
public static final double PM_FOOD_COST_PER_PERSON=8.00;
```

d. Instance variables. The "get method" for a boolean variable is usually called an "is" method. For a boolean variable called flagOn, the is method would be called isFlagOn().

```
private boolean amService;     with is and set methods
private boolean pmService;     with is and set methods
private double food;
private double foodTax;
private double foodAndTaxAmount;
```

e. Constructors.

   1) A null constructor that receives no parameters and has no statements.

   2) A constructor that receives seven parameters:

```
int reservationNumber
int seats
int numberOfDays
double dayRatePerSeat
double taxRate
boolean amService
boolean pmService
```

   This constructor passes the first five parameters to super. The last two should be passed to their set methods.

   3) A constructor that receives six parameters:

```
int reservationNumber
int seats
int numberOfDays
double dayRatePerSeat
boolean amService
boolean pmService
```

   This constructor calls the constructor that receives seven parameters and passes:

```
the first four parameters
RoomReservation82.DEFAULT_TAX_RATE
the last two parameters
```

f. A private void method called calculateAmounts.

   1) A food service order can be for AM only, PM only, or both. Use the two booleans, amService and pmService, to determine whether the daily cost per person is 0.00, 8.00, 9.00, or 17.00. Then multiply the daily cost per person by seats and numberOfDays.

   2) The foodTax is the product of multiplying food times the tax rate from the parent class RoomReservation82.

   3) The foodAndTaxAmount is the sum of food and FoodTax.

g. A public void method called printOneReservation that calls calculateAmounts and super.printOneReservation, then prints additional lines with the food charges and tax.

SOLUTIONS

E91.java

```
public class E91 {
 public static void main (String[] args) {
  MySuper parent = new MySuper ( 91 );
  System.out.println ("MySuper has " + parent.getFirst() );

  MySub child1 = new MySub ( 11, 12 );
  System.out.println ("child1 has " +
      child1.getFirst() + " and " + child1.getSecond() );

  MySub child2 = new MySub ( 22 );
  System.out.println ("child2 has " +
      child2.getFirst() + " and " + child2.getSecond() );
 }
}
```

MySuper.java

```
public class MySuper {
    private int first;

    public MySuper (int first) {
        this.first=first;
    }

    public int getFirst () {
        return first;
    }
}
```

MySub.java

```
public class MySub extends MySuper {
    private int second;

    public MySub (int first, int second) {
        super (first);
        this.second=second;
    }
    public MySub (int i) {
        this (i, i+1);
    }

    public int getSecond () {
        return second;
    }
}
```

Result, E91.java

```
MySuper has 91
child1 has 11 and 12
child2 has 22 and 23
```

FoodService.java

```
public class FoodService extends RoomReservation82 {

    public static final double AM_FOOD_COST_PER_PERSON=9.00;
    public static final double PM_FOOD_COST_PER_PERSON=8.00;

    private boolean amService;
    private boolean pmService;

    private double food;
    private double foodTax;
    private double foodAndTaxAmount;

    public FoodService () {
    }

    public FoodService (
        int reservationNumber, int seats, int numberOfDays,
        double dayRatePerSeat, double taxRate,
        boolean amService, boolean pmService
        ) {
        super (reservationNumber, seats, numberOfDays,
            dayRatePerSeat, taxRate);
        setAmService (amService);
        setPmService (pmService);
    }

    public FoodService (
        int reservationNumber, int seats, int numberOfDays,
        double dayRatePerSeat,
        boolean amService, boolean pmService
        ) {
        this (reservationNumber, seats, numberOfDays,
          dayRatePerSeat, RoomReservation82.DEFAULT_TAX_RATE,
          amService, pmService);
    }

    private void calculateAmounts () {
        double perDay = 0.0;
        if (isAmService() ) {
            perDay = AM_FOOD_COST_PER_PERSON;
        }
        if (isPmService() ) {
            perDay = perDay + PM_FOOD_COST_PER_PERSON;
        }
        food = getSeats() * getNumberOfDays() * perDay;
        foodTax = food * getTaxRate();
        foodAndTaxAmount = food + foodTax;
    }

    public void printOneReservation () {
        calculateAmounts();
        super.printOneReservation();
        System.out.println ("**Food Charges:\n" +
            "  Food before tax: " + food + "\n" +
            "  Tax:             " + foodTax + "\n" +
            "  Food and tax:    " + foodAndTaxAmount + "\n");
    }

    public boolean isAmService () {
        return amService;
    }
    public void setAmService (boolean amService) {
        this.amService = amService;
    }

    public boolean isPmService () {
        return pmService;
    }
    public void setPmService (boolean pmService) {
        this.pmService = pmService;
    }
}
```

Result, E92.java

```
Reservation: 130614
Number of seats: 12
Number of days: 5
Room amount before tax: 2700.0
Tax at 7.249999999999999%: 195.75
Final total: 2895.75

**Food Charges:
  Food before tax: 1020.0
  Tax:             73.94999999999999
  Food and tax:    1093.95

Reservation: 130782
Number of seats: 14
Number of days: 3
Room amount before tax: 1470.0
Tax at 7.249999999999999%: 106.57499999999999
Final total: 1576.575

**Food Charges:
  Food before tax: 378.0
  Tax:             27.404999999999998
  Food and tax:    405.405
```

OPTIONAL: EVERY METHOD RECEIVES A COPY OF this, DEBUG

P914.java

```
1   public class P914 {
2       public static void main (String[] args) {
3           Child c = new Child (1, 2, 3, 4);
4           c.printReport();
5       }
6   }
```

Parent.java

```
1   public class Parent {
2       private int p1, p2, pSum;
3
4       public Parent (int p1, int p2) {
5           this.p1 = p1;
6           this.p2 = p2;
7       }
8       public void calculateSum () {  //if method is private in
9           pSum = p1 * p2;            //Parent only, or in both,
10          System.out.println ("p-calc=" + this); //Parent line
11      }                              //13 calls Parent line 8
12      public void printReport () {
13          calculateSum ();
14          System.out.println ("p-print=" + this +
15          ", p1=" + p1 + ",p2=" + p2 + ",pSum=" + pSum);
16      }
17  }
```

Child.java

```
1   public class Child extends Parent {
2       private int c1, c2, cSum;
3
4       public Child (int p1, int p2, int c1, int c2) {
5           super (p1, p2);
6           this.c1 = c1;
7           this.c2 = c2;
8       }
9       public void calculateSum () {
10          cSum = c1 + c2;
11          System.out.println ("c-calc=" + this);
12      }
13      public void printReport () {
15          super.printReport();
15          calculateSum();
16          System.out.println ("c-print=" + this +
17          ", c1=" + c1 + ",c2=" + c2 + ",cSum=" + cSum);
18      }
19  }
```

Result, P914.java

```
c-calc=Child@c5f9fe
p-print=Child@c5f9fe, p1=1,p2=2,pSum=0  ---pSum wasn't calculated
c-calc=Child@c5f9fe                        because Parent line 13
c-print=Child@c5f9fe, c1=3,c2=4,cSum=7     called Child line 9
```

UNIT 10: ARRAYS

Upon completion of this unit, students should be able to:

1. Create and make use of array references, and single-dimension arrays and their individual elements.
2. Determine the number of elements in an array by using its length attribute.
3. Use the System.arraycopy() method to copy the elements from one array to another.
4. Explain the difference between arrays of basic data types and arrays of objects of class types.

ARRAY DECLARATIONS

1. An array is stored in the heap as an object and requires the use of a reference to hold its element type and symbolic address in the heap.

2. Arrays have "special support" in Java:

   a. They are created without the use of a classname.

   b. You may code an array declaration with a block that contains the element values for the array. The compiler will allocate the same number of elements as values. The new operator will be called by the compiler.

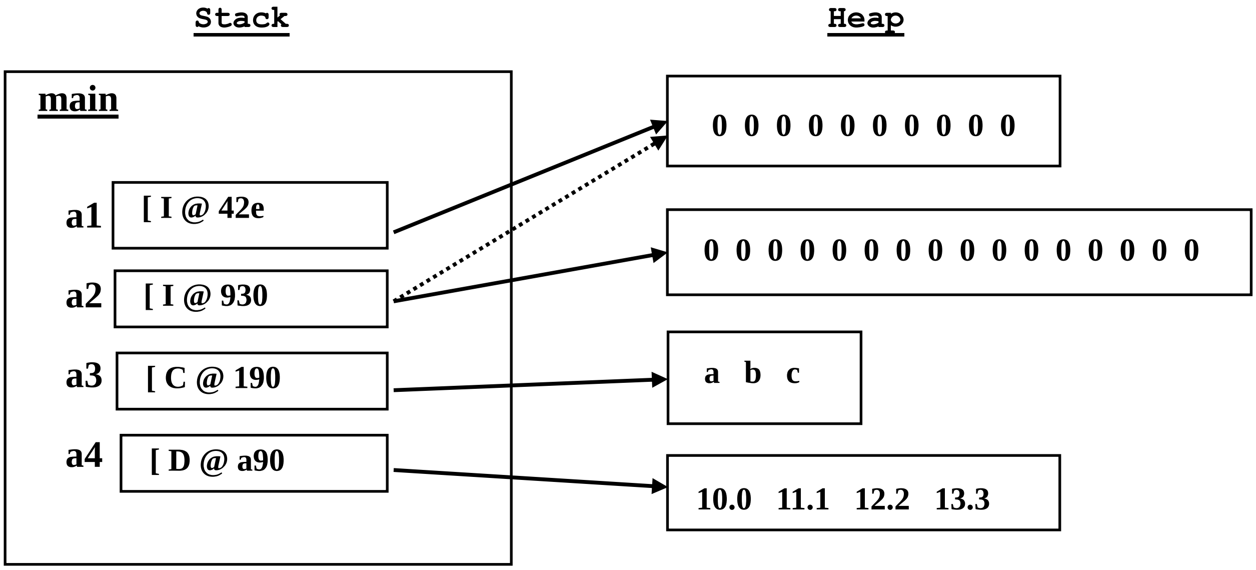
3. Array elements that you do not initialize have all bits zero.

4. Declaring an array reference does not create an array. You must also declare the array, either with the new operator or with a block of values.

5. The number of elements in an array is specified when the array object is allocated, and is not changeable. The Vector class and classes of the Collections Framework create arrays with dynamically changeable size.

6. If two references point to the same array, either one of them can be used to access or modify it.

7. If two references point to arrays with the same type of elements (such as int, double, etc.), if one reference is assigned to the other, afterward both references point to the same array, and the other array has one fewer reference.



ARRAY DECLARATIONS, EXAMPLE

P1003.java

```
public class P1003 {
    public static void main (String[] args) {

        int i = 5;                      //basic type variable

        int[] a1;                       //reference only
        a1 = new int[10];               //a1 references array

        int[] a2 = new int[16];         //reference and array

        char[] a3 = {'a', 'b', 'c'};
        double[] a4 = {10.0, 11.1, 12.2, 13.3};

        System.out.println ("i=" + i + ", a1=" + a1 +
            ", a2=" + a2 + ", a3=" + a3 + ", a4=" + a4);

        for (i=0; i<3; i++) {
            System.out.println (a3[i]);    //element notation
        }

        a2 = a1;      //16-int array is garbage-collected
                      //reference count of 10-int array is 2
        System.out.println ("a1="+a1 + ", a2="+a2);

        a2 = null;    //reference count of 10-int array is 1
    }
}
```

Result, P1003.java

```
i=5, a1=[I@42e816, a2=[I@9304b1, a3=[C@190d11, a4=[D@a90653
a
b
c
a1=[I@42e816, a2=[I@42e816
```

===

1. The values in an array are called elements. All elements in an array must be of the same data type.

2. Array elements are used like variables to contain values, and can be used in any expression valid for their data type.

3. The identifier of an element is the array name and a subscript (aka index) enclosed in [ ] square brackets.

4. A subscript must be a non-negative integer. The initial element in an array is subscript 0. The last element's subscript is the number of elements minus 1.

arrayname.length

P1004.java

```
public class P1004 {
    public static void main (String[] args) {

        int[] a1 = new int[3];
        int[] a2 = {20,21,22,23,24};

        System.out.println ("a1 length is " + a1.length);

/*1*/   if (a1.length < a2.length) {
            for (int i=0; i<a1.length; i++) {
                a2[i] = a1[i];
            }
        }

/*2*/   for (int i=0; i < a2.length; i++)
            System.out.print ("a2["+i+"]=" + a2[i]+"  ");
        System.out.println ();
    }
}
```

Result, P1004.java

```
a1 length is 3
a2[0]=0  a2[1]=0  a2[2]=0  a2[3]=23  a2[4]=24
```

==

1. Each array object automatically has a length variable called arrayname.length which is final and cannot be changed.

2. If the elements of an array are not initialized in their declaration, they automatically contain all zero bits, (integers contain 0; floating point variables contain 0.0, etc.).

3. The length variable should be used in loops to prevent subscripts from going beyond the end of the array. An ArrayIndexOutOfBoundsException occurs if a subscript goes out of bounds.

System.arraycopy()

P1005.java

```
public class P1005 {
    public static void main (String[] args) {

        int i;
        int[] a1={10,11,12,13,14,15,16,17,18,19};
        int[] a2={20,21};

        if (a1.length >= a2.length) {

            System.arraycopy (a2, 0, a1, 0, a2.length);

            System.arraycopy (a1, 2, a1, 4, 5);

            for (i=0; i < a1.length; i++)
                System.out.print (a1[i]+"  ");
            System.out.println ();

            for (i=0; i < a2.length; i++)
                System.out.print (a2[i]+"  ");
            System.out.println ();
        }
    }
}
```

Result, P1005.java

```
20  21  12  13  12  13  14  15  16  19
20  21
```

==

1. A simple loop can individually copy element values from one array to another, if element types are the same.

2. The class java.lang.System defines the method arraycopy() that copies elements from one array to another.

3. Overlapping elements can be copied within the same array.

4. Five arguments must be passed to System.arraycopy():

    a. sourceArrayName
    b. subscriptOfFirstElementToBeCopied
    c. destinationArrayName
    d. subscriptOfFirstElementToBeOverwritten
    e. numberOfElementsToBeCopied

ARRAYS OF OBJECTS

P1006.java

```
public class P1006 {
    public static void main (String[] args) {

        boolean[] bArray = {true, false, true};
        for (int i=0; i < bArray.length; i++) {
            System.out.print (i + "=" + bArray[i] + ", ");
        }

        String[] sArray = {"CA", null, "TX"};
        for (int i=0; i < sArray.length; i++) {
            if (sArray[i] == null) {
                continue;
            }
            System.out.print (i + "=" + sArray[i] + ", ");
        }
        System.out.println ("\n" + bArray + "   " + sArray);
    }
}
```

Result, P1006.java

```
0=true, 1=false, 2=true, 0=CA, 2=TX,
[Z@10948bd   [Ljava.lang.String;@8697ce
```

```
=================================================================

[Z@10948bd            ________________________
bArray--------------> |       |       |      |
                      | true  | false | true |
                      |_______|_______|______|

[Ljava.lang.String;@8697ce_____________________________________
sArray-------------> |                                         |
                     |  __sArray[0]___sArray[1]___sArray[2]__  |
                     |  |           |           |           |  |
                     |  |  String@  |  String@  |  String@  |  |
                     |  |  12ab     |  0000     |  34cd     |  |
                     |  |___________|___________|___________|  |
                     |______/_______________________\__________|
                           /                         \
     _____________________/____       ________________\_________
     |        _____________   |       |        _____________   |
     |  this | String@12ab |  |       |  this | String@34cd |  |
     |       |_____________|  |       |       |_____________|  |
     |                        |       |                        |
     |           CA           |       |           TX           |
     |________________________|       |________________________|
```

FOREACH LOOP

P1007.java

```
public class P1007 {
    public static void main (String[] args) {

        String[] sArray1={"Maine", null, "Ohio", "Alaska"};

        String[] sArray2 = new String[4];
        sArray2[0] = new String ("NY");
        sArray2[1] = new String ("NJ");
        sArray2[2] = null;
        sArray2[3] = new String ("NC");

        for (String state : sArray1) {     //loop once per
            if (state == null) {           //element in
                continue;                  //sArray1 with
            }                              //current elem's
            System.out.print (state + " ");//String reference
        }                                  //in state

        for (String abbreviation : sArray2) {
            if (abbreviation == null) {
                continue;
            }
            System.out.print (abbreviation + " ");
        }
        System.out.println ();
    }
}
```

Result, P1007.java

```
Maine  Ohio  Alaska NY NJ NC
```

1. for (dataTypeOfArrayElement nameForTempVar : arrayName)
       statement_loopBodyExecutedOnceForEachArrayElement;

2. The foreach loop uses the keyword for and must have exactly one : colon in the parentheses. Code convention is to use curly braces around the body of the loop.

3. The word after the colon is the name of the array or collection to be iterated over.

4. The first word in parentheses is the data type of the elements in the array or collection.

5. One at a time, the references in the array or collection are are copied to a temporary variable and used in one iteration of the loop. The second word in parentheses is your identifier to be used for the temporary variable.

OPTIONAL: HEAP FOR P1009.java, TWO VIEWS OF a1

```
                 ______________________________
int[][]a1------->|   _____                    |
                 |   |   |                    |
                 |   | 2 |a1.length           |
                 |   |___|                    |
                 |   _____________________    |
                 |   |         |         |    |
                 |   |  int[]  |  int[]  |    |
                 |   |reference|reference|    |
                 |   |  a1[0]  |  a1[1]  |    |
                 |   |_________|_________|    |
                 |________/___________\_______|
                         /             \
 _______________________/_______   _____\________________________
 |  _____                      |   |  _____                     |
 |  |   |a1[0].length          |   |  |   |a1[1].length         |
 |  | 4 |                      |   |  | 4 |                     |
 |  |___|                      |   |  |___|                     |
 |  _________________________  |   |  ________________________  |
 |  |     |     |     |     |  |   |  |     |     |    |     |  |
 |  | 100 | 101 | 102 | 103 |  |   |  | 110 | 111 | 112| 113 |  |
 |  |_____|_____|_____|_____|  |   |  |_____|_____|____|_____|  |
 | a1[0][0]                    |   | a1[1][0]                   |
 |_____________________________|   |____________________________|
```

```
                    _____________________________________________
                    |   _____                                   |
                    |   |_4_| a1[0].length                      |
    ____________    | _________________________________________ |
    | _______  |    | |         |         |         |         | |
a1->| |     |  |   /| |   100   |   101   |   102   |   103   | |
    | |  2  |  |  / | |_________|_________|_________|_________| |
    | |_____|  | /  |  a1[0][0]  a1[0][1]  a1[0][2]  a1[0][3]   |
    | a1.length|/   |___________________________________________|
    | _______  /
    | |     | /|    _____________________________________________
    | |a1[0]|/ |    |   _____                                   |
    | |_____|  |    |   |_4_| a1[1].length                      |
    | |     |  |    | _________________________________________ |
    | |a1[1]|--|--->| |         |         |         |         | |
    | |_____|  |    | |   110   |   111   |   112   |   113   | |
    |__________|    | |_________|_________|_________|_________| |
                    |  a1[1][0]  a1[1][1]  a1[1][2]  a1[1][3]   |
                    |___________________________________________|
```

OPTIONAL: MULTI-DIMENSIONAL ARRAYS

P1009.java

```
public class P1009 {
    public static void main (String[] args) {

        int rows, cols; //2-dimension arrays are "row-major".
                        //The first dimension is rows.
                        //The second dimension is columns.

/*1*/   int[][] a1 = {
            {100,101,102,103},   //row 0
            {110,111,112,113},   //row 1
        };

        for (rows=0; rows<2; rows++) {
            for (cols=0; cols<4; cols++) {
                System.out.print (a1[rows][cols] + "  ");
            }
            System.out.println ();
        }

/*2*/   int [][] a2 = new int[3][5];

        for (rows=0; rows < a2.length; rows++) {

            for (cols=0; cols < a2[rows].length; cols++) {

                a2[rows][cols] = 3500 + rows*10 + cols;
                System.out.print (a2[rows][cols] + "  ");

            }
            System.out.println ();

        }
    }
}
```

Result, P1009.java

```
100  101  102  103
110  111  112  113
3500  3501  3502  3503  3504
3510  3511  3512  3513  3514
3520  3521  3522  3523  3524
```

===

1. The values for array a1 can be coded indented as follows:

```
    int[][] a1 = {{100,101,102,103},{110,111,112,113},};
```

EXERCISES

1. Create a program called E101.java that contains a single-dimensional int array of 5 elements. In the declaration of the array, initialize the elements to contain the following numbers.

   ```
   1  2  4  8  16
   ```

   Display the values in the array by using a loop with System.out.print.

2. Copy E92.java and call the copy E102.java. This exercise will use RoomReservation82.java and FoodService.java without changes.

   a. Create a single-dimension FoodService array in which each element is a reference to one of your FoodService objects.

   b. In a loop, use the array elements to call the printOneReservation method for each object in the array.

SOLUTIONS

E101.java

```
public class E101 {
    public static void main (String[] args) {
        int [] a = { 1, 2, 4, 8, 16 };
        for (int i=0; i < a.length; i++) {
            System.out.print (a[i] + "  ");
        }
        System.out.println ();
    }
}
```

Result, E101.java

```
1  2  4  8  16
```

E102.java

```
public class E102 {
    public static void main (String[] args) {

    //  RoomReservation82 rr1 = new RoomReservation82();
    //  rr1.setReservationNumber (130323);
    //  rr1.setSeats (12);
    //  rr1.setNumberOfDays (5);
    //  rr1.setDayRatePerSeat (25.00);
    //  rr1.setTaxRate (0.0725);
    //  rr1.printOneReservation ();
    //
    //  RoomReservation82 rr2 = new RoomReservation82 (
    //      130444, 15, 6, 66.00, 0.21);  //invalid data
    //  rr2.printOneReservation ();
    //
    //  RoomReservation82 rr3 = new RoomReservation82 (
    //      130505, 7, 0, 24.00);         //invalid data
    //  rr3.printOneReservation ();

//first approach
        FoodService fs614 = new FoodService (
            130614, 12, 5, 45.00, 0.0725, true, true);
        FoodService fs782 = new FoodService (
            130782, 14, 3, 35.00, true, false);

        FoodService[] fsArray = new FoodService[2];
        fsArray[0] = fs614;
        fsArray[1] = fs782;

        for (FoodService elem : fsArray) {
            elem.printOneReservation ();
        }
//second approach
        FoodService[] fsArray2 = new FoodService[2];

        fsArray2[0] = new FoodService (
            130614, 12, 5, 45.00, 0.0725, true, true);
        fsArray2[1] = new FoodService (
            130782, 14, 3, 35.00, true, false);

        for (FoodService fs : fsArray2) {
            fs.printOneReservation ();
        }
//third approach
        FoodService[] fsArray3 = new FoodService[2];

        for (int i=0; i<fsArray3.length; i++) {
            fsArray3[i] = fsArray[i];
            fsArray3[i].printOneReservation ();
        }
//fourth approach
        FoodService[] fsArray4 = {
            new FoodService (
                130614, 12, 5, 45.00, 0.0725, true, true),
            new FoodService (
                130782, 14, 3, 35.00, true, false),
        };

        fsArray4[0].printOneReservation ();
        fsArray4[1].printOneReservation ();
    }
}
```

Result, E102.java

```
Reservation: 130614
Number of seats: 12
Number of days: 5
Room amount before tax: 2700.0
Tax at 7.249999999999999%: 195.75
Final total: 2895.75

**Food Charges:
  Food before tax: 1020.0
  Tax:             73.94999999999999
  Food and tax:    1093.95

Reservation: 130782
Number of seats: 14
Number of days: 3
Room amount before tax: 1470.0
Tax at 7.249999999999999%: 106.57499999999999
Final total: 1576.575

**Food Charges:
  Food before tax: 378.0
  Tax:             27.404999999999998
  Food and tax:    405.405
```

---the program produces three more sets of this output

UNIT 11: PARENT REFERENCE TO CHILD OBJECT, RUNTIME POLYMORPHISM

Upon completion of this unit, students should be able to:

1. Use the instanceof operator to determine the class type of an object.

2. Code and instantiate classes that make use of inheritance and runtime polymorphism.

3. Use abstract classes and abstract methods to facilitate runtime polymorphism.

PARENT REFERENCE TO CHILD OBJECT

P1102.java

```
1   class I {
2       private int i;
3       public I (int i) { this.i = i; }
4       public int getI () { return i; }
5   }
6   class J extends I {
7       private int j;
8       public J (int i, int j) { super(i); this.j=j; }
9       public int getSum () { return getI() + j; }
10  }
11  public class P1102 {
12      public static void main (String[] args) {
13          I ItoI = new I (1);
14          I ItoJ = new J (2, 3);
15          J JtoJ = new J (4, 5);
16          System.out.println ("ItoI=" + ItoI.getI() );
17          System.out.println ("ItoJ=" + ItoJ.getI() );
18          System.out.println ("JtoJ=" + JtoJ.getSum() );
19          System.out.println ("ItoJ=" + ItoJ.getSum() );
20      }
21  }
```

Result, attempt to compile P1102.java

```
P1102.java:19: cannot find symbol
symbol  : method getSum()
location: class I
        System.out.println ("ItoJ=" + ItoJ.getSum() );
                                          ^
1 error
```

==

1. Each reference variable is associated with a class type and a list of identifiers that would be in scope in an object of that same class type.

2. A reference of a parent type can point to an object of a subclass type because subclass objects inherit the members of the parent, and would have those resources in the object.

```
             _______________________________
ItoI------->|  this     i        getI()     |
in scope:   |  I@12ab   1        return i;  |
getI()       ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾

             ___________________________________________________
ItoJ------->|  this     i   j    getI()     getSum()            |
in scope:   |  J@34cd   2   3    return i;  return getI() + j;  |
getI()       ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾

             ___________________________________________________
JtoJ------->|  this     i   j    getI()     getSum()            |
in scope:   |  J@56ef   4   5    return i;  return getI() + j;  |
getI(), getSum()  ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾
```

THE instanceof OPERATOR

P1103.java

```
class I {
    private int i;
    public I (int i) { this.i = i; }
    public int getI () { return i; }
}
class J extends I {
    private int j;
    public J (int i, int j) { super(i); this.j=j; }
    public int getSum () { return getI() + j; }
}
public class P1103 {
    public static void main (String[] args) {

        I myI = new I (1);
        J myJ = new J (2, 3);

        if (myJ instanceof I)                     //WRONG WAY
            System.out.println ("1. myJ points to I object");
        else if (myJ instanceof J)
            System.out.println ("2. myJ points to J object");

        if (myJ instanceof J)                     //RIGHT WAY
            System.out.println ("3. myJ points to J object");
        else if (myJ instanceof I)
            System.out.println ("4. myJ points to I object");
    }
}
```

Result, P1103.java

```
1. myJ points to I object
3. myJ points to J object
```

===

1. The instanceof operator enables you to determine the class of the object that a reference variable points to.

    a. The reference variable is coded on the left (before) the instanceof operator.
    b. The class name is coded on the right (after) instanceof.

2. The terms ISA and HASA are sometimes used to describe the relationship between classes.

    a. HASA: an object "has a" reference to another object.
    b. ISA: a subclass object "is a" superclass object because the subclass object contains all of it's parent's variables and methods in the same relative location in its object space in the heap.

OPTIONAL: A SUBCLASS CAN BE USED AS SUPERCLASS TYPE, BUT ONLY SUPERCLASS IDENTIFIERS ARE IN SCOPE

1. A reference of a superclass type can refer to an object of any of its subclasses.

2. When you need a reference or object of a superclass type you may use a reference or object of its subclass instead.

    a. A method that requires a parameter of a superclass type will accept a reference to one of its subclasses.

3. The TYPE OF A REFERENCE VARIABLE determines which identifiers in the object are in scope (which ones can be accessed).

    a. Problem: If a superclass reference points to a subclass object, only identifiers of the superclass are in scope.

    b. Solution: A reference of a superclass type can be cast to the subclass type. Casting brings the identifiers of the casted subclass type into scope.

    c. Caution: At runtime, Java verifies that the casted reference in fact points to an object of the casted subclass type, and throws an Exception if it is not.

OPTIONAL: CAST A SUPERCLASS REFERENCE TO A SUBCLASS TYPE

P1105.java

```
class I {
    private int i;
    public I (int i) {
        this.i = i;
    }
    public int getI () {
        return i;
    }
}
class J extends I {
    private int j;
    public J (int i, int j) {
        super (i);
        this.j = j;
    }
    public int getJ () {
        return j;
    }
}
public class P1105 {
    public static void main (String[] args) {

        I pRef1 = new I (1);
        System.out.println ("p1="+pRef1+", i="+pRef1.getI());

        I pRef2 = new J (2, 3);
        System.out.println ("p2="+pRef2+", i="+pRef2.getI());

      //pRef2.getJ(); //cannot find symbol getJ()

        if (pRef2 instanceof J) {

            J cRef = (J) pRef2;           //cast operator (J)
            System.out.println("p=" + pRef2 + ", c=" + cRef);

            int res1 = cRef.getI()  + cRef.getJ();
            int res2 = pRef2.getI() + cRef.getJ();
            System.out.println ("1=" + res1 + ", 2=" + res2);
        }
      }
}
```

Result, P1105.java

```
p1=I@1845568, i=1
p2=J@1032cf5, i=2
p=J@1032cf5, c=J@1032cf5
1=5, 2=5
```

RUNTIME POLYMORPHISM IS TRIGGERED BY A PARENT REFERENCE TO A CHILD OBJECT, AND A CALL TO AN OVERRIDDEN METHOD

1. Runtime polymorphism means a call to an overridden method is resolved by the JVM at runtime (not by the compiler) and the JVM calls the method of the referenced object.

2. Two triggers that make the JVM do this:

   a. A superclass reference points to a subclass object, and
   b. you call an overridden method (the superclass has the method and the subclass has overridden it)

3. Runtime polymorphism lets you create a simple, consistent interface to program functionality.

   a. A superclass can define the methods common to its subclasses, and each subclass can implement its own procedure for the method as appropriate.
   b. If new subclasses are added, existing classes do not have to be modified in order to maintain a consistent interface.

4. Abstract classes facilitate runtime polymorphism by defining methods that the compiler requires subclasses to override. An abstract class cannot be instantiated, but references of an abstract class type can be used to point to objects of its concrete subclasses.

OPTIONAL

5. Two features of Java support runtime polymorphism:

   a. Method overriding, so that several methods have the same name, which represents the general functionality of the method. For example, many classes have a toString() method. Converting an Integer to a String uses a different procedure than converting a Float, but both methods have the same name.

   b. A superclass reference variable can refer to a subclass object due to their ISA relationship.

6. Methods that are private or static or final cannot be overridden, and runtime polymorphism cannot occur for them.

   a. Such a method is useful for security, because its functionality is guaranteed to occur as specified. No overriding method can change what the method does for any accidental or intentional purpose.

   b. Such a method can be useful for optimization (you may get optimized or in-line code).

RUNTIME POLYMORPHISM, EXAMPLE

P1107.java

```
1   abstract class Pet {                                 //abstract class
2       private String name;                             //can not be
3       public Pet (String n) {                          //instantiated
4           name=n;
5       }
6       public String getName() {
7           return name;
8       }
9       public abstract String getFavorite();    //abstract method
10  }                                                    //must be
11                                                       //overridden
12  class Cat extends Pet {
13      private String favoritePerch;
14      public Cat (String n, String f) {
15          super(n);
16          favoritePerch = f;
17      }
18      public String getFavorite() {           //overrides line 9
19          return favoritePerch;
20      }
21  }
22  class Dog extends Pet {
23      private String favoritePlayArea;
24      public Dog (String n, String f) {
25          super(n);
26          favoritePlayArea = f;
27      }
28      public String getFavorite() {           //overrides line 9
29          return favoritePlayArea;
30      }
31  }
32  public class P1107 {
33      public static void main (String[] args) {
34          Pet[] a = new Pet [2];          //parent refs in array
35          a[0] = new Cat ("Gert",   "windowsill");  //child obj
36          a[1] = new Dog ("Woofie", "Union Square Dogrun");
37
38          for (int i=0; i<a.length; i++) {
39              System.out.println (a[i].getName() +
40              " likes the " + a[i].getFavorite() );
41          }
42      }
43  }
```

Result, P1107.java

```
Gert likes the windowsill
Woofie likes the Union Square Dogrun
```

OPTIONAL: INTERFACES FOR STANDARDIZATION OF METHODS

1. An interface defines a group of methods that have one specific purpose, such as handling mouse clicks on a button, or handling taxes or commissions.

   a. Unrelated classes (without a superclass-subclass relationship) can implement an interface and share a defined set of methods.

   b. Each implementing class implements the methods in its own way to achieve their defined purpose.

2. An interface is like a class, but it is defined with the keyword interface.

```
public interface InterfaceName extends Inter1, Inter2 {
    datatype CONSTANT_NAME = value;
    returnvalue methodName () ;
}
```

   a. Interfaces contain only constants and abstract methods:

      1) Variables in an interface are implicitly public, static, and final, so coding these modifiers is discouraged.

      2) Methods in an interface are implicitly public and abstract, so coding these modifiers is discouraged.

3. A class that uses an interface must have an implements clause in its class header. The class then inherits the interface's constants and must implement its methods.

4. Interfaces, abstract superclasses, and multiple inheritance:

   a. An abstract class is part of the class inheritance hierarchy, but an interface is not.
   b. A class can extend only one superclass, but can implement multiple interfaces. Classes that implement a given interface are not "related" to each other.
   c. A class that implements an interface inherits only constants, not method implementations.

5. Interfaces have their own inheritance hierarchy.

6. A public interface can be accessed by any class. A non-public interface can be accessed by classes in the same package.

7. Any class with access can use an interface's constants via a qualified name, InterfaceName.CONSTANT_NAME. Implementing classes inherit the constants and can use just CONSTANT_NAME.

OPTIONAL: INTERFACE EXAMPLE

Commissions.java

```
public interface Commissions {         //Implementing classes
    int  getAgent() ;                  //must code 2 methods
    void setAgent(int agent) ;
}
```

Taxes.java

```
public interface Taxes {               //Implementing classes
    double getTaxRate() ;              //must code 2 methods
    void   setTaxRate(double taxRate) ;
}
```

Policy.java

```
public class Policy implements Commissions, Taxes {

    private String policyNo;
    private int agent;
    private double taxRate;

    public Policy () {
    }
    public Policy (String policyNo) {
        setPolicyNo (policyNo);
    }

    public String getPolicyNo () {
        return policyNo;
    }
    public void setPolicyNo (String policyNo) {
       this.policyNo = policyNo;
    }
    public String toString () {
        return "Policy:policyNo=" + policyNo;
    }

    public int    getAgent   () { return 0; }  //method stubs
    public void   setAgent   (int agent) {}
    public double getTaxRate () { return 0.0; }
    public void   setTaxRate (double taxRate) {}
}
```

P1109.java

```
public class P1109 {
    public static void main (String[] args) {
        Policy p = new Policy ( "1109" );
        System.out.println (p);
    }
}
```

Result, P1109.java

```
Policy:policyNo=1109
```

READING EXERCISE

1. Read program E111.java. Describe what the output would be and how runtime polymorphism is implemented. The answers are below.

   ANSWERS

   ```
   Gert likes the windowsill
   Woofie likes the Union Square Dogrun
   Finney likes the dried flies
   ```

   a. The Pet abstract method getFavorite() is overridden by each subclass.

   b. In main, each array element is a Pet reference that points to an object of a subclass of Pet.

   c. When getFavorite() is called on line 50, the JVM calls the method from the class of the object pointed to (Cat, Dog, or Fish) rather than the class of the reference variable (Pet).

E111.java

```java
abstract class Pet {           //5 classes in one source file
    private String name;
    public Pet (String n) {
        name=n;
    }
    public String getName() {
        return name;
    }
    public abstract String getFavorite() ;
}
class Cat extends Pet {
    private String favoritePerch;
    public Cat (String n, String f) {
        super(n);
        favoritePerch = f;
    }
    public String getFavorite() {
        return favoritePerch;
    }
}
class Dog extends Pet {
    private String favoritePlayArea;
    public Dog (String n, String f) {
        super(n);
        favoritePlayArea = f;
    }
    public String getFavorite() {
        return favoritePlayArea;
    }
}
class Fish extends Pet {
    private String favoriteFood;
    public Fish (String n, String f) {
        super(n);
        favoriteFood = f;
    }
    public String getFavorite() {
        return favoriteFood;
    }
}
public class E111 {
    public static void main (String[] args) {
        Pet[] a = {
            new Cat ("Gert", "windowsill"),
            new Dog ("Woofie", "Union Square Dogrun"),
            new Fish ("Finney", "dried flies")
        };
        for (int i=0; i<a.length; i++) {
            System.out.println (a[i].getName() +
            " likes the " + a[i].getFavorite() );
        }
    }
}
```

EXERCISES

2. Copy E102.java and call the copy CaseStudy.java. This exercise will use RoomReservation82.java and FoodService.java without changes.

    a. Create a single-dimension array of RoomReservation82 type and populate it with RoomReservation82 and FoodService objects (at least one of each class).

    b. Use a loop to traverse the array and call the method printOneReservation for each object.

    c. Does runtime polymorphism occur when you call the method printOneReservation? For which RoomReservation82 or FoodService objects? How can you know?

SOLUTIONS

CaseStudy.java

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

        RoomReservation82[] rrArray =          //parent refs
            new RoomReservation82[5];          //in the array

        rrArray[0] = new RoomReservation82();  //3 parent obj
            rrArray[0].setReservationNumber (130323);
            rrArray[0].setSeats (12);
            rrArray[0].setNumberOfDays (5);
            rrArray[0].setDayRatePerSeat (25.00);
            rrArray[0].setTaxRate (0.0725);

        rrArray[1] = new RoomReservation82 (
            130444, 15, 6, 66.00, 0.21);  //invalid data

        rrArray[2] = new RoomReservation82 (
            130505, 7, 0, 24.00);         //invalid data

        rrArray[3] = new FoodService (    //2 child objects
            130614, 12, 5, 45.00, 0.0725, true, true);

        rrArray[4] = new FoodService (
            130782, 14, 3, 35.00, true, false);

        System.out.println ("\n");
        for (RoomReservation82 obj : rrArray) {
            if (obj != null) {                 //call the
                obj.printOneReservation();     //overridden
            }                                  //method
        }
    }
}
```

Result, CaseStudy.java

```
Invalid seats 15, will be set to 12             <--error lines
Invalid numberOfDays 6, will be set to 5           should contain
Invalid dayRatePerSeat 66.0, will be set to 25.0   the key field
Invalid taxRate 0.21, will be set to 0.0725        to identify
Invalid seats 7, will be set to 12                 their record
Invalid numberOfDays 0, will be set to 5
Invalid dayRatePerSeat 24.0, will be set to 25.0

                                                <--2 blank lines
Reservation: 130323
Number of seats: 12
Number of days: 5
```

```
Room amount before tax: 1500.0
Tax at 7.249999999999999%: 108.74999999999999
Final total: 1608.75

Reservation: 130444
Number of seats: 12
Number of days: 5
Room amount before tax: 1500.0
Tax at 7.249999999999999%: 108.74999999999999
Final total: 1608.75

Reservation: 130505
Number of seats: 12
Number of days: 5
Room amount before tax: 1500.0
Tax at 7.249999999999999%: 108.74999999999999
Final total: 1608.75

Reservation: 130614
Number of seats: 12
Number of days: 5
Room amount before tax: 2700.0
Tax at 7.249999999999999%: 195.75
Final total: 2895.75

**Food Charges:
  Food before tax: 1020.0
  Tax:             73.94999999999999
  Food and tax:    1093.95

Reservation: 130782
Number of seats: 14
Number of days: 3
Room amount before tax: 1470.0
Tax at 7.249999999999999%: 106.57499999999999
Final total: 1576.575

**Food Charges:
  Food before tax: 378.0
  Tax:             27.404999999999998
  Food and tax:    405.405
```

UNIT 12: PACKAGES, IMPORT, final

Upon completion of this unit, students should be able to:

1. Briefly explain the purpose of packages and the import directive, and code programs that use them.

2. Briefly explain how the keyword final affects variables, methods, and classes.

COMPILATION UNITS AND PACKAGES

1. A compilation unit consists of the source code for one or more classes and interfaces. A compilation unit:

    a. Is usually one source file
    b. May contain a maximum of one public class or interface
    c. Must belong to exactly one package

2. A package is a group of related compilation units, and is usually implemented as a directory. Packages are used to organize classes and to limit namespaces.

3. Examples of packages are java.lang and java.util.

4. The package statement is a compiler directive that specifies the name and location of a compilation unit's package.

    a. The package statement must be the first statement in a compilation unit preceded only by whitespace and comments.

    b. If no package statement is specified, the compilation unit belongs to the default "unnamed" package, which is your current directory. All classes we created so far are in the unnamed package.

5. If a class is not public, it can be referenced only by other classes in the same package.

6. A class can refer to a public class or interface in a different package in two ways:

    a. Qualify the class or interface name with its package name and a period. For example, InputStream in java.io is java.io.InputStream.

    b. Import the package.

7. If you work on a commandline rather than an IDE such as Eclipse, place your main class in the top directory of your project, and do all your work while you are in the top directory. TO AVOID COMPILE AND EXECUTION ERRORS WITH RELATIVE PATHNAMES, ALWAYS STAY IN YOUR TOP DIRECTORY.

PACKAGES, EXAMPLE

```
current package
    P1203.java
    animals package
        Cat.java
        Dog.java
```

P1203.java in current package

```
public class P1203 {
    public static void main (String[] args) {

        animals.Cat c = new animals.Cat ("Liberty", "desk");
        animals.Dog d = new animals.Dog ("Cisco", "hall");
        System.out.println (c + "   " + d);
    }
 }
```

Cat.java in package animals

```
package animals;
public class Cat {
     private String name;
     private String favoritePerch;
     public Cat (String n, String p) {
         name = n;
         favoritePerch = p;
     }
     public String toString() {
        return "Cat:" + name + "," + favoritePerch;
    }
}
```

Dog.java in package animals

```
package animals;
public class Dog {
    private String name;
    private String favoritePlayArea;
    public Dog (String n, String p) {
        name = n;
        favoritePlayArea = p;
    }
    public String toString() {
        return "Dog:" + name + "," + favoritePlayArea;
    }
}
```

Result, P1203.java

```
Cat:Liberty,desk   Dog:Cisco,hall
```

THE import STATEMENT

1. The import statement is a compiler directive that enables you to refer to a public class or interface in a different package by its simple name, without qualifying the name.

2. If your code uses a class that is not in the current package, the import statement:

   a. gives javac permission to look in a different package
   b. tells javac what package to look in
   c. tells javac where the package is

3. Importing a package does not cause the compiler to read any class or interface definitions, which occurs only if your code makes use of a class or interface.

4. The import statement must be located after the package statement if there is one, and before any class or interface declaration.

5. The scope of the import is from its location to the end of its compilation unit.

6. Two ways to code import:

   a.
   ```
   import packagename.ClassName;
   //ClassName or packagename.ClassName may be used
   ```

   b.
   ```
   import packagename.*;
   //javac will search packagename for classes and
   //interfaces referenced but not defined
   ```

7. Each package must be imported separately, even if their names are related, such as java.awt and java.awt.image.

8. It is an ambiguity error if javac searches the packages you specify and finds more than one class with a given name. To resolve the ambiguity, import the specific classname that you wish to use, as shown in 6.a. above, or use a fully qualified name each time you refer to the class.

THE import STATEMENT, EXAMPLE

P1205.java in current package

```
import animals.Cat;                                    //new
import animals.Dog;                                    //new

public class P1205 {
    public static void main (String[] args) {

        Cat c = new Cat ("Liberty", "desk");     //different
        Dog d = new Dog ("Cisco", "hall");       //different
        System.out.println (c + "   " + d);
    }
}
```

Cat.java in animals package

```
package animals;

public class Cat {
     private String name;
     private String favoritePerch;

     public Cat (String n, String p) {
         name = n;
         favoritePerch = p;
     }
     public String toString() {
        return "Cat:" + name + "," + favoritePerch;
    }
}
```

Dog.java in animals package

```
package animals;

public class Dog {
    private String name;
    private String favoritePlayArea;

    public Dog (String n, String p) {
        name = n;
        favoritePlayArea = p;
    }
    public String toString() {
        return "Dog:" + name + "," + favoritePlayArea;
    }
}
```

Result, P1205.java

```
Cat:Liberty,desk   Dog:Cisco,hall
```

final CLASSES, METHODS, AND VARIABLES

P1206.java

```
class I {
    public static final int USEFUL_NUM = 123;
    private int i;
    public I (int i) {
        this.i=i;
    }
    public final int getTot () {
        return i;
    }
}

final class J extends I {
    private int j;
    public J (int i, int j) {
        super(i);
        this.j=j;
    }
    //public int getTot () {  }  **Can't override**
}

public class P1206 {
    public static void main (String[] args) {

        I objI = new I (1);
        J objJ = new J (10, 20);

        System.out.println ("useful=" + I.USEFUL_NUM +
            ", objI.getTot=" + objI.getTot() +
            ", objJ.getTot=" + objJ.getTot() );
    }
}
```

Result, P1206.java

```
useful=123, objI.getTot=1, objJ.getTot=10
```

1. If the keyword final is applied to a class, the class can have no subclasses, and all methods in the class are implicitly final.

2. If the keyword final is applied to a method, the method can not be overridden. This enables the compiler to resolve calls during compile time or to use inline bytecode, either of which can result in faster execution.

3. If the keyword final is applied to a variable, the variable can be assigned a value only one time, whether in the variable declaration or later in a procedural statement.

EXERCISES

1. Use the javadoc 1.8 to answer the following questions. You can do an internet search on "javadoc 1.8 api" to find it.

   a. What package is the class FileInputStream in?

   b. What is the superclass of FileInputStream?

   c. Is FileInputStream's superclass abstract?

   d. Does FileInputStream implement any interfaces?

   e. How many known (that is, listed in the javadoc) subclasses does FileInputStream's superclass have?

   f. What package is the class Number in?

   g. What is the superclass of Number?

   h. How many known subclasses does the Number class have?

   j. Does Number implement any interfaces?

   i. Do all of Number's subclasses have to override all of Number's methods?

   k. What package is the Object class in?

   l. What class in the javadoc can you look at to determine the class type of <u>out</u> which we use in the expression System.out.println?

SOLUTIONS

1. Use the javadoc 1.8 to answer the following questions. You can do an internet search on "javadoc 1.8 api" to find it.

   a. What package is the class FileInputStream in?
   java.io

   b. What is the superclass of FileInputStream?
   InputStream

   c. Is FileInputStream's superclass abstract?
   Yes

   d. Does FileInputStream implement any interfaces?
   No

   e. How many known (that is, listed in the javadoc) subclasses does FileInputStream's superclass have?
   9

   f. What package is the class Number in?
   java.lang

   g. What is the superclass of Number?
   Object

   h. How many known subclasses does the Number class have?
   10

   j. Does Number implement any interfaces?
   Yes, Serializable

   i. Do all of Number's subclasses have to override all of Number's methods?
   No, because byteValue() and shortValue() are not abstract

   k. What package is the Object class in?
   java.lang

   l. What class in the javadoc can you look at to determine the class type of out which we use in the expression System.out.println?
   Look at System for the variable out. The Field Summary shows that it is type PrintStream.

UNIT 13: STRING, STRINGBUFFER, STRINGBUILDER

Upon completion of this unit, students should be able to:

1. Briefly describe what a String is, and how String differs from StringBuffer and StringBuilder.

2. Locate documentation for the String, StringBuffer, and StringBuilder classes and their methods.

3. Declare, initialize, and assign values to String, StringBuffer, and StringBuilder objects.

4. Work with the contents of String objects by using String static and instance methods such as String.valueOf, charAt, compareTo, concat, endsWith, equals, equalsIgnoreCase, length, replace, startsWith, substring, toLowerCase, and toUpperCase.

5. Use the toString method to print a String version of the data members in a class.

STRING LITERALS, REVIEW

P1302.java

```java
public class P1302 {
    public static void main (String[] args) {

        System.out.println (13 + "" + 2 +
        ", strings in \", "   +   "chars in '");
    }
}
```

Result, P1302.java

```
132, strings in ", chars in '
```

1. The String class may be the most frequently used class in the Java API. The String class contains variables and methods used to represent and manipulate strings.

2. A string is a sequence of zero or more characters stored in an object of type String.

3. A String literal is compiled into a String object, and an internal, compiler-created reference points to the object.

4. The \ backslash character in a String has to be coded as the escape sequence \\.

5. Unicodes can be used in Strings to designate any character except newline and return, which are coded as \n and \r.

6. A String literal must be coded on one line of source code. There is no continuation from line to line, but multiple Strings can be concatenated into one String by using the concatenation operator + plus.

7. A String is NOT a char array, and the following will NOT compile: `char[] Str = "abc";`

STRING OBJECT DECLARATION AND INITIALIZATION

P1303.java

```
public class P1303 {
    public static void main (String[] args) {

        String s1;
        s1 = new String ("April in Paris");

        String s2 = new String ("Christmas in Moscow");

        String s3 = "Memorial Day in Prospect Park";

        s3 = "Thanksgiving in Honolulu";

        s3 = s1;

        System.out.print (s1 + "\n" + s2 + "\n" + s3);
    }
}
```

Result, P1303.java

```
April in Paris
Christmas in Moscow
April in Paris
```

===

1. A String variable is a reference variable that points to an object of type String.

2. A String reference can be initialized in the declaration in two ways:

   a. String identifier = new String ("value");

   b. String identifier = "value";

3. One String reference can be assigned to another. After the assignment, both String references point to the same object. If the reference count for a String object goes down to zero, the object will be deallocated by the garbage collector.

4. After a String object is created, its contents cannot be changed. For example, if you concatenate to a String, your String will be garbage-collected and a new String created.

5. The StringBuffer and StringBuilder classes should be used for operations that modify strings (such as appending, inserting, and concatenating) so that garbage collection is not unnecessarily burdened.

STRING METHODS

P1304.java

```
public class P1304 {
  public static void main (String[] args) {

    System.out.print ("1. " + String.valueOf (13.04) );

    String p="Paris";
    String m="Moscow";
    boolean b = p.equals(m);
    if (b) System.out.println (", 2. p=m");

    if (p.equals("paris")) System.out.print (", 3");
    if ("Paris".equals(p)) System.out.print (", 4");
    if (p.startsWith("Pa")) System.out.print (", 5");
    if (p.endsWith("is")) System.out.print (", 6");
    if (p.equalsIgnoreCase("pARIS"))System.out.print (", 7");

    int i = p.compareTo (m);
    System.out.println ("\n8. negative-0-positive=" + i);

    System.out.print ("9. length=" + p.length());
    System.out.print (", 10. lower=" + p.toLowerCase());
    System.out.print (", 11. upper=" + p.toUpperCase());
    System.out.print ("\n12. concat=" + p.concat("!"));
    System.out.print (", 13. substr=" + p.substring(1,3));
    System.out.print (", 14. " + m.replace('o','O') );
    System.out.print (", 15. charAt=" + p.charAt(2) );
  }
}
```

Result, P1304.java

```
1. 13.04, 4, 5, 6, 7
8. negative-0-positive=3
9. length=5, 10. lower=paris, 11. upper=PARIS
12. concat=Paris!, 13. substr=ar, 14. MOscOw, 15. charAt=r
```

===

1. The compareTo method accepts one parameter, compares the instance String to the parameter String char by char from left to right, and returns an int that is:
   a. Negative if the instance is lower than the parameter.
   b. Zero if the instance is the same as the parameter.
   c. Positive if the instance is higher than the parameter.

2. The substring instance method treats the instance String as a char array indexed starting from 0. The method accepts two parameters:
   a. Index of the first char to be returned.
   b. Index of the NEXT CHAR AFTER THE LAST CHAR to be returned.

StringBuffer, StringBuilder

P1305.java

```java
public class P1305 {
    private static int varI    = 123;
    private static double varD = 4.5;

    public static void main (String[] args) {
        System.out.println ( useStringBuffer() );
        System.out.println ( useStringBuilder() );
    }

    public static String useStringBuffer () {
        return new StringBuffer("StringBuffer:")
            .append("varI=").append(varI)
            .append(",varD=").append(varD)
            .toString();
    }

    public static String useStringBuilder () {
        return new StringBuilder("StringBuilder:")
            .append("varI=")
            .append(varI)
            .append(",varD=")
            .append(varD)
            .toString();
    }
}
```

Result, P1305.java

```
StringBuffer:varI=123,varD=4.5
StringBuilder:varI=123,varD=4.5
```

1. String objects are unchangeable. When you concatenate strings, you are creating a new string object, deallocating old ones, and increasing the work of the garbage collector.

2. StringBuffer is a thread-safe class that allows modification of the data contained in the object. Append and insert are the most commonly used StringBuffer methods.

3. StringBuilder is not thread-safe, which may make it more efficient. It allows the same modifications to the data as StringBuffer. The StringBuilder class was added to the Java API in Java 5.

public String toString() FROM THE Object CLASS

1. The top class in the Java class hierarchy is Object in the package java.lang.

2. Every class in Java inherits some methods from Object, including the toString method.

3. The toString method of Object should be overridden by every class so that an object of the class can represent its important data members in one String.

4. When System.out.println or System.out.print must print a reference to an object, they call the toString() method of the object the reference is pointing to.

5. toString() must be public. This is because the method in Object is public, and an overriding method in a subclass cannot narrow the access of the overridden superclass method.

OPTIONAL NOTES

6. The code below shows how to ask an object what class type it was constructed from.

   a. In Y.java on line 4, the Object method getClass returns a reference to a Class object.

   b. Y.java on line 5, the Class method getSimpleName returns the name of the underlying class as a String.

X.java

```
1   public class X {
2       public static void main (String[] args) {
3
4           Y ref = new Y ();
5           System.out.println ("classname is " + ref );
6       }
7   }
```

Y.java

```
1   public class Y {
2
3       public String toString() {
4           Class refClassObj = this.getClass ();
5           String myClassName = refClassObj.getSimpleName();
6           return myClassName;
7       }
8   }
```

Result, X.java

```
classname is Y
```

public String toString() FROM Object CLASS, EXAMPLE

P1307.java

```
class A {
    private int a;
    public A (int a) {
        this.a=a;
    }
    public String toString() {
        return String.valueOf(a);
    }
}
class B extends A {
    private int b;
    public B (int a, int b) {
        super(a);
        this.b=b;
    }
    public String toString() {
        return "B:"+super.toString()+","+String.valueOf(b);
    }
}
class C extends B {
    private int c;
    public C (int a, int b, int c) {
        super(a, b);
        this.c=c;
    }
}
class D {
    private int d;
    public D (int d) {
        this.d=d;
    }
}
public class P1307 {
    public static void main (String[] args) {
        A myA = new A (1);
        B myB = new B (10, 20);
        C myC = new C (100, 200, 300);
        D myD = new D (123);
        System.out.println (myA+"  "+myB+"  "+myC+"  "+myD);
    }
}
```

Result, P1307.java

```
1  B:10,20  B:100,200  D@1bccd400
```

THE HEAP FOR P1309.java

```
                 ab1234___________________________________
String[]args---->|                                       |
String[]@ab1234  |   A String array is implemented as    |
                 |   an array of references to String    |
                 |   objects.                            |
                 |   _____                               |
                 |   |   |                               |
                 |   | 2 |args.length                    |
                 |   |___|                               |
                 |                                       |
                 |   args[0]_________args[1]__________   |
                 |   |               |               |   |
                 |   |    String     |    String     |   |
                 |   |   reference   |   reference   |   |
                 |   |               |               |   |
                 |   | String@cd5678 | String@ef9abc |   |
                 |   |_______________|_______________|   |
                 |        /                   \          |
                 |_______/_____________________\_________|
                        /                       \
                       /                         \
                      /                           \
cd5678_______________/_________   ef9abc___________\_____________
|                             |   |                             |
|  String object              |   |  String object              |
|                             |   |                             |
|  args[0].length() returns 11|   |  args[1].length() returns 7 |
|  _______________            |   |  ___________                |
|  |             |            |   |  |         |                |
|  | commandline |            |   |  | example |                |
|  |_____________|            |   |  |_________|                |
|                             |   |                             |
|  charAt()                   |   |  charAt()                   |
|  compareTo()                |   |  compareTo()                |
|  concat()                   |   |  concat()                   |
|  endsWith()                 |   |  endsWith()                 |
|  equals()                   |   |  equals()                   |
|  length()                   |   |  length()                   |
|  replace()                  |   |  replace()                  |
|  startsWith()               |   |  startsWith()               |
|  substring()                |   |  substring()                |
|  toLowerCase()              |   |  toLowerCase()              |
|  toUpperCase()              |   |  toUpperCase()              |
|_____________________________|   |_____________________________|
```

COMMANDLINE ARGUMENTS AND String[] args

P1309.java

```java
public class P1309 {
   public static void main (String[] args) {

      int i;
      int numElementsInArray = args.length;      /*variable*/

      for (i=0; i<numElementsInArray; i++) {
         System.out.println (i + ". " + args[i]);
      }

      if (numElementsInArray > 0) {
         int numCharsInString=args[0].length();    /*method*/
         System.out.println ("strlen=" + numCharsInString);
      }
   }
}
```

Result, P1309.java with 2 arguments: commandline example

```
0. commandline
1. example
strlen=11
```

===

1. When you execute your program on a commandline, the words following the name of your program are stored as elements of a String array called (by convention) args.

2. Your programming environment dictates how you can enter commandline arguments.

3. When an array contains basic types, the basic variables are in the array object.

4. When an array contains class types, the array contains references, and the objects of the array are located in the heap wherever the JVM finds space for them.

5. A "pure java" program should follow POSIX conventions for commandline options and arguments.

6. Entering commandline arguments in Eclipse is covered in Appendix E. In UNIX and DOS windows:

   a. UNIX:
   ```
   $ javac P1309.java
   $ java P1309 commandline example
   ```

   b. DOS:
   ```
   C:\myjava> javac P1309.java
   C:\myjava> java P1309 commandline example
   ```

EXERCISES

1. Create a program called E131.java that declares three String references and initializes them to point to String objects as follows:

String	initialize the
Reference	object to contain
first	your first name
last	your last name
wish	a wish you have

   a. Concatenate the three Strings into one String called me.

   b. Display the length of the String me.

   c. Display the number of times the letter e appears in the String me.

   d. Display the String me with the characters in reverse order.

   e. Display the String me in all upper case.

   f. Display the String me in all lower case.

2. Revise program P1302.java from this unit to create a new program called E132.java. Replace the arguments in the System.out.println twice, once using StringBuffer and once using StringBuilder.

SOLUTIONS

E131.java

```java
public class E131 {
    public static void main (String[] args) {

        String first="Teresa";
        String last="Hommel";
        String wish="Peace";
        String me;

        //a.  Concatenate the 3 strings into one called me.
        me = String.valueOf (first);
        me = me.concat(last);
        me = me.concat(wish);
        System.out.println ("a. concat=" + me );

        //b.  Display the length of the string me.
        System.out.println ("b. length=" + me.length());

        //c.  Display number of e letters in the string me.
        int i, count=0;
        for (i=0; i<me.length(); i++)
            if (me.charAt(i) == 'e') count ++;
        System.out.println ("c. count of e=" + count);

        //d.  Display the string me backward.
        for (i=me.length()-1; i>=0; i--)
            System.out.print (me.charAt(i) );
        System.out.println ();

        //e.  Display the string me in all upper case.
        System.out.println ("e. upper=" + me.toUpperCase());

        //f.  Display the string me in all lower case.
        System.out.println ("f. lower=" + me.toLowerCase());
    }
}
```

Result, E131.java

```
a. concat=TeresaHommelPeace
b. length=17
c. count of e=5
ecaePlemmoHasereT
e. upper=TERESAHOMMELPEACE
f. lower=teresahommelpeace
```

E132.java

```
//P1302.java revised
public class E132 {
    public static void main (String[] args) {

        //System.out.println (13 + "" + 2 +
        //", strings in \", "   +   "chars in '");

        System.out.println (
            new StringBuffer("13")
            .append("2")
            .append(", strings in \", ")
            .append("chars in '")
            .toString()
            );

        System.out.println (
            new StringBuilder("13")
            .append("2")
            .append(", strings in \", ")
            .append("chars in '")
            .toString()
            );
    }
}
```

Result, E132.java

```
132, strings in ", chars in '
132, strings in ", chars in '
```

UNIT 14: WRAPPER CLASSES

Upon completion of this unit, students should be able to:

1. Briefly explain the purpose of wrapper classes.

2. Use the methods and variables available in the wrapper classes to process numeric and character data.

OVERVIEW OF THE WRAPPER CLASSES

1. The wrapper classes are useful because:

    a. They provide a way to encapsulate the value of any basic data type into an object.

    b. They provide many static (aka class) methods to perform commonly-needed tasks with variables of the basic data types.

    c. They provide constants which define the maximum and minimum values that can be stored in variables of the basic data types, such as Integer.MAX_VALUE and Integer.MIN_VALUE.

2. Wrapper classes are used to satisfy the requirements of methods that require their parameters to be passed as objects, and classes that require their data to be stored in objects.

3. The Collections Framework and the Reflection API may require the use of the wrapper classes because basic type variables, if needed, must be wrapped in objects.

4. The wrapper classes provide many useful methods.

    a. valueOf() is a static method. When a String contains characters that signify a number, valueOf() can convert the String to the numeric contents of a wrapper class object.

    b. toString() has both static and instance versions that can convert the numeric contents of a wrapper class object to a String.

5. All wrapper classes are defined in java.lang. The wrapper classes are:

    a. Number

    b. Byte, Short, Integer, Long, Float, and Double

    c. Character

    d. Boolean

    e. Void

THE ABSTRACT SUPERCLASS Number

1. Number is an abstract superclass whose subclasses provide object wrappers for the numeric basic data types: byte, short, int, long, float, and double.

2. The Number class defines six instance methods.

    a. byteValue(), returns the value of its object as a byte.

    b. shortValue(), returns the value of its object as a short.

    c. intValue(), returns the value of its object as an int.

    d. longValue(), returns the value of its object as a long.

    e. floatValue(), returns the value of its object as a float.

    f. doubleValue(), returns the value of its object as a double.

3. Because all the subclasses of Number implement all of these methods, the numeric value stored in an object of any Number subclass can be retrieved as a value of any basic numeric data type. However, if the data type of the object is not the same as the return value, rounding may occur.

4. Number inherits the following instance methods from its superclass Object. The subclasses of Number implement them.

    a. public boolean equals (Object obj) returns true if the instance object is the same class type and contains the same value as the parameter object.

    b. public String toString() returns a String representation of the instance object.

Integer

1. Integer is a subclass of java.lang.Number.

2. Integer provides constants, constructors, class methods, and instance methods for processing int values.

3. The Integer constructor is overloaded so that it can accept an int or a String as an argument.

4. Class methods:

    a. int parseInt(String s)
    i. accepts a String
    ii. returns an int with the value represented by the characters in the String.

    b. Integer valueOf(String s)
    i. accepts a String
    ii. returns an Integer object with the value represented by the characters in the String.

    c. String toBinaryString(int i);
    String toHexString(int i);
    String toOctalString(int i);
    i. accept an int
    ii. return a String representing the int value as characters of the specified number system.

    d. String toString(int i)
    i. accepts an int
    ii. returns a String with characters representing the decimal value of the int parameter.

5. Instance methods:

    a. String toString()
    i. accepts no arguments
    ii. returns a String with characters representing the value of this Integer object.

    b. boolean equals(Object o)
    i. accepts an object to be compared with this object
    ii. returns true if the argument is an Integer object containing the same value as this Integer object.

    c. byteValue(), shortValue(), intValue(), longValue(), floatValue(), doubleValue(),
    i. accept no arguments
    ii. return a variable of the requested basic data type containing the value of this Integer object.

Integer EXAMPLE

P1405.java
```
public class P1405 {
    public static void main (String[] args) {

/*1*/   int i = 12;                               //i has 12
        Integer ref = new Integer (i);            //ref to 12
        p ("1. i=" + i + ", r=" + ref);

/*2*/   long n = 34L;
        if (n>=Integer.MIN_VALUE && n<=Integer.MAX_VALUE) {
            ref = new Integer ( (int)n );         //ref to 34
            p ("2. fits in Integer: " + ref);
        }

/*3*/   String stringNum = "56";
        int i = Integer.parseInt (stringNum);     //i has 56
        ref  =  Integer.valueOf  (stringNum);     //ref to 56
        p ("3. from String to i=" + i + " or ref=" + ref);

/*4*/   stringNum = ref.toString();         //instance method
        stringNum = Integer.toString (78);     //class method
        p ("4. from object or int to String=" + stringNum);

/*5*/   float f = ref.floatValue();
        p ("5. from object to any basic type=" + f);

/*6*/   String b, o, h;
        b = Integer.toBinaryString (90);
        o = Integer.toOctalString  (90);
        h = Integer.toHexString    (90);
        p ("6. d=90, b=" + b + ", o=" + o + ", h=" + h);

/*7*/   Integer ref2 = new Integer (1234);
        if ( ref.equals(ref2) )
            p ("7. " + ref + " = " + ref2);
    }

    static void p (String s) {
        System.out.println (s);
    }
}
```

Result, P1405.java
```
1. i=12, r=12
2. fits in Integer: 34
3. from String to i=56 or ref=56
4. from object or int to String=78
5. from object to any basic type=56.0
6. d=90, b=1011010, o=132, h=5a
```

Character

1. Character is a class in the java.lang package.

2. Character provides constants, constructors, class methods, and instance methods for processing char values.

3. The Character constructor accepts a char value.

4. Class methods that accept a char argument and return a boolean true or false based on the category of the char in the Unicode character set:

    a. isDefined(), true if the char has a defined meaning

    b. isDigit(), true if the char is defined as a digit

    c. isLetter(), true if the char is defined as a letter

    d. isLetterOrDigit(), true if the char is defined as a letter or digit

    e. isLowerCase(), true if the char is defined as lowercase

    f. isUpperCase(), true if the char is defined as uppercase

5. Class methods that accept a char argument and return the same char in upper or lower case if it is a letter, or the same char unchanged if the argument is not a letter:

    a. toLowerCase()

    b. toUpperCase()

6. Instance methods:

    a. char charValue()
        i. accepts no arguments
        ii. returns the char value of this object.

    b. boolean equals(Object o)
        i. accepts an object to be compared with this object
        ii. returns boolean true if the argument is a Character object containing the same value as this Character object.

    c. String toString()
        i. accepts no arguments
        ii. returns a String of length one containing the value of this Character object.

Character EXAMPLE

P1407.java

```
public class P1407 {
    public static void main (String[] args) {

    System.out.println("1. "+Character.isDigit('a') );

    System.out.println("2. "+Character.isLetter('a') );

    System.out.println("3. "+Character.isLetterOrDigit('a'));

    System.out.println("4. "+Character.isLowerCase('a') );

    System.out.println("5. "+Character.isUpperCase('a') );

    System.out.println("6. "+
        Character.isLowerCase( Character.toLowerCase('A') ));


    char c = Character.toUpperCase('a');
    System.out.println ("7. " + c);


    Character obj1 = new Character ('1');
    Character obj2 = new Character ('2');

    c = obj1.charValue();
    System.out.println ("8. " + c);

    if (obj1.equals(obj2)) System.out.println ("9. true");
    else System.out.println ("10. false");

    String s = obj1.toString();
    System.out.println ("11. " + s);
    }
}
```

Result, P1407.java

```
1. false
2. true
3. true
4. true
5. false
6. true
7. A
8. 1
10. false
11. 1
```

EXERCISES

1. Create a program called E141.java in which the main method contains four Strings:

```
"123"
"1x3"
"123.4"
"1 2 3"
```

   a. One at a time, the main method passes each String to a method called isInt that determines whether or not the String could be an int and returns true if it can.

   b. The isInt method uses these two requirements for an int:

      1) must consist entirely of digits.
      2) must be nine digits or fewer. (See page 3.04 for the minimum and maximum int value.)

   c. In main, if isInt returns true, main passes the String to a method called convertAndDisplayInt that converts the String to an int and to an Integer, and displays the numeric value in each one with an appropriate message.

SOLUTIONS

E141.java

```java
public class E141 {
    public static void main (String[] args) {
        String[] a = {
            "123",
            "1x3",
            "123.4",
            "1 2 3"
        };
        for (String s : a) {
            if (isInt(s)) {
                convertAndDisplayInt (s);
            }
        }
    }

    public static boolean isInt (String s) {
        for (int i=0; i<s.length(); i++) {
            if (! Character.isDigit(s.charAt(i)) ) {
                return false;
            }
        }
        if (s.length() < 10) {
            return true;
        } else {
            return false;
        }
    }

    public static void convertAndDisplayInt (String s) {
        int i = Integer.parseInt(s);
        Integer obj = new Integer (s);
        System.out.println ("int="+i + ", Integer="+obj);
    }
}
```

Results, E141.java

```
int=123, Integer=123
```

UNIT 15: EXCEPTIONS

Upon completion of this unit, students should be able to:

1. Briefly describe the purpose of exception handling, and how to create and handle an exception.

2. Use the keywords throw, throws, try, catch, and finally to handle exceptions.

3. Briefly describe the organization of standard exceptions, and which ones are required to be handled.

4. Display a stack trace of method calls.

EXCEPTIONS

1. An exception is a predefined unusual condition or violation of a rule, such as ArrayIndexOutOfBoundsException.

2. The purpose of exception handling is to allow or require the program to handle or recover from predictable unusual conditions, rather than letting the program prematurely exit.

3. When an exception occurs in a called method, the method "throws" an Exception object to its caller. This lets the caller detect and handle exceptions in the called method.

4. If the caller handles the exception, program execution resumes with the code following the list of catch clauses in which the exception was handled.

5. If the caller does NOT handle the exception, the exception propagates up to the next higher calling method, and so on until the exception is passed up to the Java Virtual Machine, which then terminates program execution.

6. Each different exception is predefined by its own class, which must be a subclass of the Exception class in java.lang.

7. In an exception class, the constructor need not do anything, but one constructor should accept a String argument to allow a descriptive message to be passed.

8. When the exceptional condition occurs, the code must create and throw an object of the exception class. For example:

    a. throw new MyOwnException ();
    b. throw new MyOwnException ("some useful info");

9. A method that can throw an exception must declare this possibility in a throws clause in its header. For example:

    ```
    static void myMethod() throws MyOwnException {
    ```

10. Throwing an exception causes flow of control to return to the method's caller. Unlike the return statement, throw does not go back to the next action following the call, but rather to the appropriate exception handler in the caller.

11. One way to handle an exception is via try {} catch {}. The try block encloses the call to the method that may throw the exception. Each catch clause specifies the exception it handles in parentheses, and how to handle it in curly braces.

12. Both try and catch require the use of curly braces.

EXCEPTION FLOW OF CONTROL, EXAMPLE

MyException.java

```
1   public class MyException extends Exception {
2       public MyException () {
3       }
4       public MyException (String s) {
5           super(s);
6       }
7   }
```

P1503.java

```
1   public class P1503 {
2       public static void main (String[]a) throws MyException {
3
4           System.out.println ("1. main");
5
6           try {
7               throwMethod ('a');
8           } catch (MyException e) {
9               System.out.println ("3. catch, e=" + e);
10          }
11
12          System.out.println ("4. main");
13          throwMethod ('c');
14          throwMethod ('b');
15      }
16
17      static void throwMethod (char ch) throws MyException {
18
19          System.out.println ("2. method called with " + ch);
20
21          if (ch == 'a')
22              throw new MyException ("a helpful message");
23          if (ch == 'b')
24              throw new MyException ();
25      }
26  }
```

Result, P1503.java

```
1. main
2. method called with a
3. catch, e=MyException: a helpful message
4. main
2. method called with c
2. method called with b
Exception in thread "main" MyException
        at P1503.throwMethod(P1503.java:24)
        at P1503.main(P1503.java:14)
```

OPTIONAL: WAYS TO HANDLE EXCEPTIONS

1. An exception can be completely handled in the catch clause, that is, your code can "fix" the problem.

2. An exception can be allowed to propagate up to the next higher calling method without being caught. The only code required for this is a throws clause in this method's header.

3. An exception can be partially handled and then rethrown in the catch clause. This requires a throws clause in this method's header.

4. An exception can be replaced by another exception. To throw a different exception, instead of lines 7, 13, and 19 on the facing page, you could code:

```
7   } catch (MyDifferentException m) {

13  public static void sub1() throws MyDifferentException {

19  throw new MyDifferentException();
```

5. An exception can be wrapped in another exception.

   a. To wrap the exception in another exception, instead of lines 7, 13, and 19 on the facing page, you could code:

```
7   } catch (MyDiffException m) {

13  public static void sub1() throws MyDiffException {

19  throw new MyDiffException( m );
```

   b. The wrapper exception class needs a constructor that receives an Exception, which enables it to receive an object of any exception class, or the specific exception that it will receive. For example:

```
1   public class MyDiffException extends Exception {
2       public MyDiffException () {
3       }
4       public MyDiffException (String s) {
5           super(s);
6       }
7       public MyDiffException (Exception e) {
8           super(e);
9      }
10  }
```

   c. The output of line 8 would be:
   5. catch, m=MyDiffException: MyException

OPTIONAL: WAYS TO HANDLE EXCEPTIONS, EXAMPLE

MyException.java

```
public class MyException extends Exception {
    public MyException () {
    }
    public MyException (String s) {
        super(s);
    }
}
```

P1505.java

```
public class P1505 {

    public static void main (String[] args) {
        try {
            System.out.println ("1. main before sub1");
            sub1();
        } catch (MyException m) {
            System.out.println ("5. catch, m=" + m );
        }
        System.out.println ("6. main after sub1");
    }

    public static void sub1() throws MyException {
        System.out.println ("2. sub1 before sub2");
        try {
            sub2();
        } catch (MyException m) {
            System.out.println("4. sub1 caught m from sub2");
            throw m;
        }
    }

    public static void sub2() throws MyException {
        sub3();
    }

    public static void sub3() throws MyException {
        System.out.println ("3. sub3");
        throw new MyException ();
    }
}
```

Result, P1505.java

```
1. main before sub1
2. sub1 before sub2
3. sub3
4. sub1 caught m from sub2
5. catch, m=MyException
6. main after sub1
```

finally CLAUSE

AException.java

```
public class AException extends Exception {
}
```

BException.java

```
public class BException extends Exception {
}
```

P1506.java

```
public class P1506 {
    public static void main (String[] args) {

        try {
            sub ( 0 );
        } catch (AException ae) {
            System.out.print ("ae=" + ae + ", ");
        } catch (BException be) {
            System.out.print ("be=" + be + ", ");
        } finally {
            System.out.print ("finally, ");
        }
        System.out.println ("after try-catch");
    }

    public static void sub (int i)
        throws AException, BException {
        if ( i == 0 )
            throw new AException ();
        if ( i == 1 )
            throw new BException ();
        return;
    }
}
```

Result, P1506.java

```
ae=AException, finally, after try-catch
```

1. A try must have at least one catch or finally clause.

2. The finally clause requires a set of curlies.

3. A finally clause is executed before flow of control leaves the try, whether or not an exception occurred. It is used for closing files and network connections, and freeing resources.

4. If System.exit occurs in the try, finally is not done. If a return occurs in the try, finally is done first.

STANDARD EXCEPTIONS IN java.lang

1. Many exceptions can be "thrown" by the Java Virtual Machine during program execution, or by many of the pre-defined methods in the Java API.

2. The java.lang package contains the Throwable class and its two subclasses, Exception and Error.

3. Part of the Exception class hierarchy:
    I. Object
        A. Throwable
            1. Error
            2. Exception
                a. ClassNotFoundException
                b. IllegalAccessException
                c. InstantiationException
                d. NoSuchFieldException
                e. NoSuchMethodException
                f. RuntimeException
                    1) ArithmeticException
                    2) ClassCastException
                    3) IllegalArgumentException
                        a) IllegalThreadStateException
                        b) NumberFormatException
                    4) IndexOutOfBoundsException
                        a) ArrayIndexOutOfBoundsException
                        b) StringIndexOutOfBoundsException
                    5) NegativeArraySizeException
                    6) NullPointerException

4. A method that can throw an exception must acknowledge that it might do so via a throws clause in the method header, except that runtime exceptions do not have to be acknowledged because they might occur in any method.

    a. An exception that must be acknowledged via a try-catch or a throws clause is called a checked exception.

    b. An exception that does not have to be acknowledged is called an unchecked exception.

5. Errors in the Error class are typically thrown by the class loader or the Java Virtual Machine. Normally your program would not throw one of these errors. If a method does throw one of them, it does not have to acknowledge it. Most errors cannot be handled, and will cause your program to exit.

6. If you call a method that throws an exception (other than a runtime exception), the compiler will require you to code the method call within a try block, OR put the appropriate throws clause in your own method header.

printStackTrace()

MyException.java

```
1   public class MyException extends Exception {
2   }
```

P1508.java

```
1   public class P1508 {
2       public static void main (String[] args) {
3           try {
4               sub1();
5           } catch (MyException m) {
6               m.printStackTrace();
7           }
8       }
9       public static void sub1() throws MyException {
10          sub2();
11      }
12      public static void sub2() throws MyException {
13          sub3();
14      }
15      public static void sub3() throws MyException {
16          throw new MyException ();
17      }
18  }
```

Result, P1508.java

```
MyException
     at P1508.sub3(P1508.java:16)
     at P1508.sub2(P1508.java:13)
     at P1508.sub1(P1508.java:10)
     at P1508.main(P1508.java:4)
```

1. When an exception could come from more than one sequence of called methods, displaying a stack trace can help debugging.

2. printStackTrace() is defined in the Throwable class and is inherited by all its subclasses. Two ways to call it are:

   a.
   ```
   try {
       //methods that throw exceptions
   } catch (AnException ae) {
       ae.printStackTrace();
   }
   ```

   b.
   ```
   new Throwable().printStackTrace(); //returns only the
                                      //line where it was
                                      //coded
   ```

EXERCISES

1. Copy E141.java and call the copy E151.java.

    a. Create an Exception class called BadDataException.java with two constructors, one null and one that receives a String parameter and passes it to super.

    In E151.java

    b. In the method isInt, if the String is not all digits or if it is longer than 9 characters, throw a BadDataException with an appropriate String describing why the data is invalid, and the String containing the bad data.

    c. In the main method, use a finally clause to print the number of valid Strings that could be converted to int or Integer.

SOLUTIONS

BadDataException.java

```
public class BadDataException extends Exception {
    public BadDataException () {
    }
    public BadDataException (String s) {
        super(s);
    }
}
```

E151.java

```
public class E151 {
    public static void main (String[] args) {

        int validCount = 0;
        String[] a = {"123", "1x3", "123.4", "1 2 3"};

        /* version 1, finds only the first invalid String
        try {
            for (String s : a) {
                if (isInt(s)) {
                    convertAndDisplayInt (s);
                    validCount++;
                }
            }
        } catch (BadDataException bde) {
            bde.printStackTrace();
        } finally {
            System.out.println ("valid=" + validCount);
        }
        */

        // version 2, find and count all invalid Strings
        int invalidCount = 0;
        for (String s : a) {
            try {
                if (isInt(s)) {
                    convertAndDisplayInt (s);
                    validCount++;
                }
            } catch (BadDataException bde) {
                bde.printStackTrace();
                invalidCount++;
            } finally {
                System.out.println ("valid=" + validCount +
                    ", invalid=" + invalidCount + "\n");
            }
        }
    }

40      public static boolean isInt (String s)
41      throws BadDataException {
42          for (int i=0; i<s.length(); i++) {
43              if (! Character.isDigit(s.charAt(i)) ) {
44                  throw new BadDataException (
45                  "bad int, non-digit character=" + s);
46              }
47          }
48          if (s.length() < 10) {
49              return true;
50          } else {
51              throw new BadDataException (
52              "bad int, more than 9 digits=" + s);
53          }
54      }
55
56      public static void convertAndDisplayInt (String s) {
57          int i = Integer.parseInt(s);
58          Integer obj = new Integer (s);
59          System.out.println ("int="+i + ", Integer="+obj);
60      }
61  }
```

Result, E151.java, version 1 which finds only the first invalid

```
int=123, Integer=123
BadDataException: bad int, non-digit character=1x3
     at E151.isInt(E151.java:32)
     at E151.main(E151.java:17)
valid=1
```

Result, E151.java, version 2

```
int=123, Integer=123
valid=1, invalid=0

BadDataException: bad int, non-digit character=1x3
     at E151.isInt(E151.java:43)
     at E151.main(E151.java:26)
valid=1, invalid=1

BadDataException: bad int, non-digit character=123.4
     at E151.isInt(E151.java:43)
     at E151.main(E151.java:26)
valid=1, invalid=2

BadDataException: bad int, non-digit character=1 2 3
     at E151.isInt(E151.java:43)
     at E151.main(E151.java:26)
valid=1, invalid=3
```

(blank)

UNIT 16: java.io, File, BYTE STREAMS, CHARACTER STREAMS

Upon completion of this unit, students should be able to:

1. Briefly describe the purpose of the File class, and use File objects to store filenames in a platform independent way, obtain information about files or directories, and perform other functions with files and directories.

2. Briefly describe the difference between byte streams and character streams.

3. State which classes in java.io handle byte and character streams.

4. Create a program that reads and writes ordinary disk files using byte streams and/or character streams.

5. State which classes in java.io are node streams and wrapper streams. Wrap a node stream in a buffering wrapper.

6. Use an InputStreamReader to read an InputStream, and an OutputStreamWriter to write an OutputStream, to bridge between byte and character streams.

A File OBJECT STORES A FILENAME

P1602.java

```java
import java.io.File;
public class P1602 {
    public static void main (String[] args) {

        //f1.txt and f2.txt exist, but sub does not
        File f1  = new File ("f1.txt");
        File f2  = new File ("d:/myjava", "f2.txt");
        File dir = new File ("d:\\myjava\\sub");
        File f3  = new File (dir, "f3.txt");

        boolean r = f1.canRead ();
        boolean w = f1.canWrite ();
        boolean e = f1.exists ();
        System.out.println ("1. " +r+ ", " +w+ ", " +e);

        boolean d = f1.isDirectory ();
        boolean f = f1.isFile ();
        long len = f1.length ();
        System.out.println("2. " +d+ ", " +f+ ", len=" +len);

        boolean del = f1.delete ();
        e = f1.exists ();
        System.out.println ("3. " +del+ ", " +e);
    }
}
```

Result, P1602.java

```
1. true, true, true
2. false, true, len=20
3. true, false
```

1. An object of the File class holds the name of a disk file or directory, and can be used to obtain information about the file or directory. The File class has three constructors:

    a. public File (String path)
    b. public File (String path, String name)
    c. public File (File dir, String name)

2. The delete() method can delete a file or empty directory, and returns true if the file or directory is deleted.

3. In pathnames, the forward slash can be used even in DOS windows. To code a backslash, use the escape sequence \\.

java.io, BYTE AND CHARACTER STREAMS, STANDARD STREAMS

1. The java.io package contains classes that perform input and output operations with disk files as well as other sources and destinations of streams of data.

2. A stream is a flow of bytes or characters that can be read as input into a program from a source, or written as output from a program to a destination.

3. The source or destination of a stream can be a disk file, keyboard or monitor display unit (console screen), internal buffer, or network socket. The stream concept allows the java.io classes to handle input and output easily in spite of the differences between different sources and destinations.

4. Byte streams can be read and written by the subclasses of the abstract classes InputStream and OutputStream.

5. At the hardware level, all input and output is done with bytes, and binary data is byte-oriented. However, to support internationalization, character streams of Unicode characters can be read and written by the subclasses of the abstract classes Reader and Writer. (Byte stream classes that handle Unicode characters do so by using solely the least significant 8 bits, which does not always represent the Unicode character correctly.)

6. java.lang.System defines three public static final constants that are references to objects representing standard input, standard output, and standard error.

constant	represents	object of
System.in	standard input	java.io.InputStream
System.out	standard output	java.io.PrintStream
System.err	standard error	java.io.PrintStream

HIERARCHY OF BYTE STREAM CLASSES

Inheritance Relationships	Ctor Parameters
A. InputStream (abstract class)	
1. ByteArrayInputStream	byte[] buf
2. FileInputStream	String filename, File f, fildes
3. FilterInputStream	InputStream
a. BufferedInputStream	InputStream
b. DataInputStream	InputStream
c. LineNumberInputStream (deprecated)	
d. PushbackInputStream	InputStream
4. ObjectInputStream	InputStream
5. PipedInputStream	PipedOutputStream
6. SequenceInputStream	InputStream
7. StringBufferInputStream (deprecated)	
B. OutputStream (abstract class)	
1. ByteArrayOutputStream	uses internal buffer
2. FileOutputStream	String filename, File f, fildes
3. FilterOutputStream	OutputStream
a. BufferedOutputStream	OutputStream
b. DataOutputStream	OutputStream
c. PrintStream	deprecated constructors
4. ObjectOutputStream	OutputStream
5. PipedOutputStream	PipedInputStream

==

1. Subclasses of InputStream are commonly called InputStreams. Subclasses of OutputStream are commonly called OutputStreams.

2. InputStreams and OutputStreams that connect to a source or destination of data are called node streams.

3. An InputStream that accepts a constructor argument that is a reference to an InputStream object, or an OutputStream that accepts a constructor argument that is a reference to an OutputStream, are called wrapper streams. Example:

   a. If you instantiate a FileInputStream, and pass its reference to the constructor of BufferedInputStream, your BufferedInputStream object can perform buffered input with data that the FileInputStream reads. This is called "wrapping" (or chaining or decorating) a FileInputStream in a BufferedInputStream.

4. PrintStream constructors were deprecated in Java 1.1 in favor of PrintWriters (covered later in this unit) because PrintStreams don't handle Unicodes well. Because System.out and System.err are PrintStreams, the methods of PrintStream are not deprecated, but you should not create new PrintStream objects.

SOME METHODS DEFINED IN InputStream AND OutputStream

1. InputStream is an abstract class that defines input methods to be implemented by concrete subclasses.

   a. int read(), reads one byte and returns it in the least significant byte of an int, or returns an int containing -1 for end of file.

   b. int read(byte[] buf), reads up to buf.length bytes into buf and returns how many bytes were read or -1 for end of file.

   c. int read(byte[] buf, int offset, int numBytes), reads up to numBytes bytes into buf starting at offset, and returns how many bytes were read or -1 for end of file. The initial byte of buf is offset zero.

   d. void close(), closes the input source. Subsequent reads from it will cause an IOException.

2. OutputStream is an abstract class that defines output methods to be implemented by concrete subclasses.

   a. void write(int b), writes the byte portion of the int b. The byte portion is the least significant eight bits of the int.

   b. void write(byte[] buf), writes buf.length bytes from buf.

   c. void write(byte[] buf, int offset, int numBytes), writes numBytes from buf[offset]. The initial byte of buf is offset zero.

   d. void flush(), flushes (writes) the output buffer.

   e. void close(), closes the output stream. Subsequent writes to it will cause an IOException.

3. Most input and output methods can throw exceptions.

FileInputStream, FileOutputStream, COPY ONE BYTE AT A TIME

P1606.java

```
1   import java.io.InputStream;
2   import java.io.FileInputStream;
3   import java.io.OutputStream;
4   import java.io.FileOutputStream;
5
6   public class P1606 {
7       public static void main (String[] a) throws Exception {
8
9           InputStream fis = new FileInputStream ("indata");
10          OutputStream fos = new FileOutputStream ("out");
11
12          int tot=0;
13          int inputHolder;
14
15          while (  (inputHolder = fis.read() ) != -1) {
16              fos.write (inputHolder);
17              tot++;
18          }
19
20          fis.close();
21          fos.close();
22          System.out.println ("Number of bytes copied=" + tot);
23      }
24  }
```

indata

```
This is a data file to be copied.
```

out (before)

```
Pre-existing data should be backed up.
```

Result, P1606.java

```
Number of bytes copied=34                     ---UNIX file length
```

out (after)

```
This is a data file to be copied.
```

===

1. Disk files contain bytes. Objects of FileInputStream and FileOutputStream are commonly used to read and write files.

2. When an output file is created, any pre-existing file with the same name is deleted. This occurs in the program above when the new FileOutputStream object is created on line 10.

3. The read method above accepts no parameters and returns int. It reads one byte and returns it in the least significant (rightmost) byte of an int, or returns an int containing -1 to signify end of file.

FILE STREAMS, COPY ONE RECORD AT A TIME

P1607.java

```java
import java.io.InputStream;
import java.io.FileInputStream;
import java.io.OutputStream;
import java.io.FileOutputStream;
public class P1607 {
    public static void main (String[] a) throws Exception {
        int numBytesRead=0;
        int byteCount=0;
        int recordCount=0;
        byte[] buf = new byte [10];
        InputStream in = new FileInputStream ("in");
        OutputStream out = new FileOutputStream ("out");

        while (  (numBytesRead = in.read(buf)) != -1) {

            //validate number of bytes read

            out.write (buf, 0, numBytesRead);
            byteCount = byteCount + numBytesRead;
            recordCount++;
        }
        in.close();
        out.close();
        System.out.println ("records=" + recordCount +
                          ", bytes=" + byteCount);
    }
}
```

in

```
aaaaa11111bbbbb22222ccccc33333ddddd44444     ---no newline at end
```

Result, P1607.java

```
records=4, bytes=40
```

out

```
aaaaa11111bbbbb22222ccccc33333ddddd44444     ---no newline at end
```

1. Record-oriented files are usually organized into fixed-length segments, not lines. In a UNIX window, if you display a file that does not end with a newline, the next shell prompt will appear on the same line with the last byte of file data.

2. The read method above accepts one parameter that is a reference to a byte array that should be the same length as your input records, and returns int. The method reads up to buf.length bytes into buf and returns how many bytes were read or -1 for end of file.

BufferedInputStream, BufferedOutputStream

P1608.java

```
import java.io.InputStream;
import java.io.FileInputStream;
import java.io.BufferedInputStream;
import java.io.OutputStream;
import java.io.FileOutputStream;
import java.io.BufferedOutputStream;
public class P1608 {

    private static final int RECORD_SIZE = 10;

    public static void main (String[] a) throws Exception {
        int numRead=0;
        byte[] buf = new byte [RECORD_SIZE];
        InputStream fis = new FileInputStream ("in");
        InputStream bis = new BufferedInputStream (fis);

        OutputStream bos = new BufferedOutputStream (
            new FileOutputStream ("out")  );

        while (  (numRead = bis.read(buf)) != -1) {
            if (numRead != RECORD_SIZE) {
                System.err.println ("EOF size error");
                System.exit (1);
            }
            bos.write (buf, 0, RECORD_SIZE);
        }

        bis.close();
        bos.close();
    }
}
```

in (before)

```
aaaaa11111bbbbb22222ccccc33333ddddd44444     ---no newline at end
```

out (after)

```
aaaaa11111bbbbb22222ccccc33333ddddd44444     ---no newline at end
```

==

1. Wrapping any InputStream (or OutputStream) object in a BufferedInputStream (or BufferedOutputStream) attaches an internal buffer to the stream, so that input (or output) operations are more efficient.

CHARACTER STREAM CLASSES

Inheritance Relationships	Ctor Parameters
A. Reader (abstract class)	
1. BufferedReader	Reader
a. LineNumberReader	Reader
2. CharArrayReader	char[] buf
3. FilterReader	Reader
a. PushbackReader	Reader
4. InputStreamReader (bridge class)	InputStream
a. FileReader	String filename, File f, fildes
5. PipedReader	PipedWriter
6. StringReader	String
B. Writer (abstract class)	
1. BufferedWriter	Writer
2. CharArrayWriter	uses internal buffer
3. FilterWriter	Writer
4. OutputStreamWriter (bridge class)	OutputStream
a. FileWriter	String filename, File f, fildes
5. PipedWriter	PipedReader
6. PrintWriter	OutputStream or Writer
7. StringWriter	uses internal buffer

1. Java classes that handle byte streams do not handle Unicode characters well.

2. To support internationalization, Java provides a second group of classes to handle streams of Unicode characters.

3. Character streams are defined by two class hierarchies descending from the abstract classes Reader and Writer.

4. The classes that accept a Reader constructor argument can wrap an object of any subclass of Reader. The classes that accept a Writer constructor argument can wrap an object of any subclass of Writer.

5. InputStreamReader and OutputStreamWriter are called "bridge classes" because they wrap an InputStream or OutputStream and convert correctly between bytes and characters. PrintWriter is another wrapper that can wrap either an OutputStream or another Writer.

SOME METHODS DEFINED IN Reader AND Writer

1. Reader is an abstract superclass that defines input methods to be implemented by concrete subclasses.

    a. int read(), reads one char and returns it in the least significant two bytes of an int, or returns -1 for end of file.

    b. int read(char[] buf), reads up to buf.length chars into buf and returns how many were read or -1 for end of file.

    c. int read(char[] buf, int offset, int numChars), reads up to numChars chars into buf[offset] and returns how many chars were read or -1 for end of file.

    d. void close(), closes the input source. Subsequent reads from it will cause an IOException.

2. Writer is an abstract superclass that defines output methods to be implemented by concrete subclasses.

    a. void write(int ch), writes the char portion of the int ch. The char portion of an int is the least significant two bytes.

    b. void write(char[] buf), writes buf.length chars.

    c. void write(char[] buf, int offset, int numChars), writes numChars from buf[offset].

    d. void write(String s), writes s.

    e. void write(String s, int offset, int numChars), writes numChars chars from s starting at offset.

    f. void flush(), flushes (writes) the output buffer.

    g. void close(), closes the output stream. Subsequent writes to it will cause an IOException.

3. Most input and output methods can throw exceptions.

FileReader, FileWriter

P1611.java

```
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.Reader;
import java.io.FileReader;
import java.io.Writer;
import java.io.FileWriter;
public class P1611 {
    public static void main (String[] args)
    throws FileNotFoundException, IOException {

        Reader fr = new FileReader ("data.txt");
        Writer fw = new FileWriter ("mycopy");

        int numRead;
        int tot=0;
        char[] buf = new char[10];

        while ((numRead=fr.read(buf)) != -1) {
            fw.write (buf, 0, numRead);
            tot = tot + numRead;
        }

        System.out.println ("Number of chars copied=" + tot);
        fr.close();
        fw.close();
    }
}
```

data.txt (before)

```
This is a data file to be copied.
```

mycopy (before)

```
Pre-existing data should be backed up.
```

Result, P1611.java

```
Number of chars copied=34                    ---UNIX file length
```

mycopy (after)

```
This is a data file to be copied.
```

1. FileReader reads bytes from a file and uses the default character encoding scheme to convert each byte to a 2-byte char. FileWriter uses the default character encoding scheme to convert each char to a byte, and writes it to a file.

2. One FileWriter constructor lets you specify appending rather than overwriting to a pre-existing file. If you try to write to a read-only file, FileWriter throws an IOException.

BufferedReader, BufferedWriter

P1612.java

```
import java.io.Reader;
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.Writer;
import java.io.FileWriter;
import java.io.BufferedWriter;
public class P1612 {
    public static void main (String[] arg) throws Exception {

        Reader fr = new FileReader ("data.txt");
        BufferedReader br = new BufferedReader (fr);
        //BufferedReader ref needed; Reader has no readLine()

        Writer bw = new BufferedWriter (
            new FileWriter ("out",true) );   //true to append
                                         //false to overwrite
        bw.write ('*');

        String s;
        while ( (s=br.readLine()) != null)
            bw.write (s, 0, s.length());  //str,start,howMany

        char[]a = {'E','N','D','.','\n'};
        bw.write (a);

        br.close();
        bw.close();
    }
}
```

data.txt

```
This is a data file to be copied.
```

out (before)

```
Pre-existing data should be backed up.
```

out (after)

```
Pre-existing data should be backed up.
*This is a data file to be copied.END.
```

===

1. For greater efficiency, any Reader or Writer may be wrapped in a BufferedReader or BufferedWriter.

2. The readLine method of BufferedReader reads a line, truncates the line separator (\n or \r or \r\n), and returns the chars as a String, or returns null at end of file.

BufferedReader, BufferedWriter, ANOTHER EXAMPLE

P1613.java

```
1   import java.io.*;
2   public class P1613 {
3       public static void main (String[] a) throws Exception {
4
5           FileReader fr = new FileReader ("data.txt");
6           BufferedReader br = new BufferedReader (fr);
7
8           FileWriter fw = new FileWriter ("mycopy");
9           BufferedWriter bw = new BufferedWriter (fw);
10          PrintWriter pw = new PrintWriter (bw);
11
12          String lineBuf;
13
14          while (  (lineBuf = br.readLine()) != null) {
15
16              pw.println ("println power and flexibility");
17              pw.write (lineBuf);
18              pw.write (System.getProperty("line.separator") );
19          }
20          br.close();
21          pw.close();
22      }
23  }
```

data.txt (before)

```
This is a data file to be copied.
```

mycopy (before did not exist, after contains 2 lines)

```
println power and flexibility
This is a data file to be copied.
```

==

1. By wrapping a Writer object in a PrintWriter, you get access to the print and println methods.

2. The following code does the same as lines 5 and 6 above.

```
         BufferedReader br = new BufferedReader (
             new FileReader ("data.txt")  );
```

3. The following code does the same as lines 8 through 10 above.

```
        PrintWriter pw = new PrintWriter (
            new BufferedWriter (new FileWriter ("mycopy") ) );
```

WRAP System.in

P1614.java

```
1   import java.io.*;
2   public class P1614 {
3       public static void main (String[] a) throws IOException {
4
5           System.out.print ("Enter a line: ");
6           System.out.flush ();
7
8           InputStreamReader i=new InputStreamReader(System.in);
9           BufferedReader br = new BufferedReader (i);
10
11          String s;
12          if ( (s=br.readLine()) == null) {
13              System.exit (1);
14          }
15          int len = s.length();
16          System.out.println ("line=" + s + ", len=" + len);
17      }
18  }
```

Result, P1614.java

```
Enter a line: when the moon comes over the mountain
line=when the moon comes over the mountain, len=37
```

Commandline execution

```
$ java P1614 < empty.txt                   ---input redirection
Enter a line:                                 with an empty file
$
```

1. To facilitate internationalization, InputStreamReader is a bridge class between bytes and chars. It can read an InputStream of bytes, and convert them to chars. To create an InputStream object that wraps System.in, use the constructor shown on line 8.

2. Lines 8 and 9 can be written in one statement:

```
        BufferedReader br = new
            BufferedReader (new InputStreamReader (System.in));
```

WRAP System.out

P1615.java

```
1   import java.io.*;
2   public class P1615 {
3       public static void main (String[] a) throws IOException {
4
5           PrintWriter pw = new PrintWriter (System.out, true);
6
7           pw.print ("Enter a line: ");
8           pw.flush();
9
10          InputStreamReader r=new InputStreamReader(System.in);
11          BufferedReader br = new BufferedReader (r);
12
13          String s;
14          if ( (s=br.readLine()) == null)
15              System.exit (1);
16
17          pw.println ("line=" + s + ", len=" + s.length());
18      }
19  }
```

Result, P1615.java

```
Enter a line: When the moon comes over the mountain
line=When the moon comes over the mountain, len=37
```

Commandline execution

```
$ java P1615 < empty.txt                     ---input redirection
Enter a line:                                   with an empty file
$
```

1. Use of System.out is recommended primarily for debugging and for illustrating the features of Java in training materials.

2. In applications that must write to the console, use a PrintWriter stream, which has the same print and println methods, but facilitates internationalization.

3. For the following constructor, if flush is true, the output stream will be flushed each time a newline is written.

   ```
   PrintWriter (OutputStream outStream, boolean flush);
   ```

4. The flush method, used on line 8 above, may be needed if the output to be printed does not end in a newline, or if there is buffereing and the buffer is not full.

BufferedInputStream WITH VARIABLE LENGTH RECORDS

P1616.java

```
import java.io.*;
public class P1616 {

    private static byte[] buf = new byte[8];

    public static void main (String[] args)throws Exception {
        int numRead = 0;
        int len = 8;
        BufferedInputStream bis = new BufferedInputStream (
            new FileInputStream ("in.txt") );

        while((numRead=bis.read(buf, 8-len,  len))!= -1){
                              //buf, offset, numBytesToRead

            for (int i=0; i<8; i++)
               System.out.print(buf[i]+"="+(char)buf[i]+" ");
            System.out.println ();

            switch (  (char)buf[0]  ) {
               case '4' : printRec(4); len=4; break;
               case '6' : printRec(6); len=6; break;
               case '8' : printRec(8); len=8; break;
            }

            System.arraycopy (buf,len,  buf,0,     8-len);
        }                  //src,start  dest,start howmany
        bis.close();
    }

    public static void printRec(int len) {
        String s = new String (buf, 0, len);
        System.out.println (s);
    }
}
```

in.txt ---32 bytes, no newline at end

```
8aaaaaaa6bbbbb4ccc6ddddd8eeeeeee
```

Result, P1616.java

```
56=8 97=a 97=a 97=a 97=a 97=a 97=a 97=a
8aaaaaaa
54=6 98=b 98=b 98=b 98=b 98=b 52=4 99=c
6bbbbb
52=4 99=c 99=c 99=c 54=6 100=d 100=d 100=d
4ccc
54=6 100=d 100=d 100=d 100=d 100=d 56=8 101=e
6ddddd
56=8 101=e 101=e 101=e 101=e 101=e 101=e 101=e
8eeeeeee
```

UNIT 17: DATES, CALENDARS, AND NUMBERS

Upon completion of this unit, students should be able to:

1. Use the Date class to represent a particular moment in time in a standard or tailorable date and time format.

2. Use the DateFormat, SimpleDateFormat, and Locale classes to format a date in various ways.

3. Use the Calendar class to convert Date information to "calendar" fields such as day of the week or year.

4. Use the NumberFormat and Locale classes to edit a number for report printing.

Date

1. The java.util.Date class is used to represent a particular moment in time in a standard or tailorable date and time format.

2. A Date object can be created with the current date and time, or any date and time for which you provide a long that contains the number of milliseconds elapsed since the first millisecond of January 1, 1970, GMT.

   a. Date myDate = new Date ();
   b. Date myDate = new Date (123456789L);

3. Get and set methods for the time:

   a. public long getTime();
   b. public long setTime (long newTime);

4. Compare the time in two Date objects:

   a. public boolean after (Date when);
   b. public boolean before (Date when);
   c. public boolean equals (Date when);

5. To calculate calendar values such as month, day of the week, or julian day of the year, use the java.util.Calendar class.

6. To parse a string that represents a date and determine what date it is, use the java.text.DateFormat class.

Date, DATES BEFORE OR AFTER, EXAMPLE

P1703.java

```
import java.util.Date;

public class P1703 {
    public static void main (String[] args) {

        Date now = new Date ();
        System.out.println ("1. now is " + now);

        long nowMS = now.getTime();
        System.out.println ("2. now in MS is " + nowMS);

        long oneDayMS = 1000 * 24 * 60 * 60;
        long tomorrowMS = nowMS + oneDayMS;

        Date tomorrow = new Date ( tomorrowMS );
        System.out.println ("3. tomorrow is " + tomorrow);

        boolean a = tomorrow.after (now);
        boolean b = now.before (tomorrow);
        boolean e = now.equals (tomorrow);
        System.out.println ("4. " + a + ", " + b + ", " + e);

        now.setTime (now.getTime() - (2*oneDayMS) );
        System.out.println ("5. two days ago was " + now);
    }
}
```

Result, P1703.java

```
1. now is Tue Jul 27 19:42:04 GMT-05:00 2010
2. now in MS is 1280277724703
3. tomorrow is Wed Jul 28 19:42:04 GMT-05:00 2010
4. true, true, false
5. two days ago was Sun Jul 25 19:42:04 GMT-05:00 2010
```

DateFormat, Locale

1. The java.text.DateFormat class enables you to format a date in various ways. This class uses the international locale provided by java.util.Locale to determine the appropriate form for the date.

2. The java.text.DateFormat instance method parse() can examine a String that represents a date, and returns a reference to a Date object for that date. A java.text.ParseException will be thrown if the String does not contain a recognizable date.

Result, P1705.java

```
1.  short: java.text.SimpleDateFormat@8629ad2d
2.  short: 7/27/10
3.  medium: Jul 27, 2010
4.  long: July 27, 2010
5.  full: Tuesday, July 27, 2010
6.  default: Jul 27, 2010
7.  time: 8:15:18 PM
8.  Thu Aug 12 00:00:00 GMT-05:00 2010
```

DateFormat, DATE PRINTOUTS, EXAMPLE

P1705.java

```
import java.util.Date;
import java.util.Locale;
import java.text.DateFormat;
import java.text.ParseException;

public class P1705 {
    public static void main (String[] args) {

        Date d = new Date ();
        Locale.setDefault (Locale.US);

        DateFormat dfS =
            DateFormat.getDateInstance (DateFormat.SHORT);
        System.out.println ("1.  short: " + dfS);
        System.out.println ("2.  short: " + dfS.format (d));

        DateFormat dfM =
            DateFormat.getDateInstance (DateFormat.MEDIUM);
        System.out.println ("3.  medium: " + dfM.format (d));

        DateFormat dfL =
            DateFormat.getDateInstance (DateFormat.LONG);
        System.out.println ("4.  long: " + dfL.format (d));

        DateFormat dfF =
            DateFormat.getDateInstance (DateFormat.FULL);
        System.out.println ("5.  full: " + dfF.format (d));

        DateFormat dfD =
            DateFormat.getDateInstance (DateFormat.DEFAULT);
        System.out.println ("6.  default: "+ dfD.format (d));


        DateFormat t = DateFormat.getTimeInstance ();
        System.out.println ("7.  time: " + t.format (d));


        String stringDate = "8/12/10";
        try {
            Date S = dfS.parse (stringDate);
            System.out.println ("8.  " + S);
        } catch (ParseException pe) {
            System.out.println ("pe=" + pe);
        }
    }
}
```

Calendar

1. The java.util.Calendar class is an abstract class with methods to convert the information in Date objects to "calendar fields", such as, for Thursday April 1, 1999, at 11:49:04 in the morning, Eastern Standard Time:

   a. YEAR=1999
   b. MONTH=3 (January is 0)
   c. WEEK_OF_YEAR=14
   d. WEEK_OF_MONTH=1
   e. DAY_OF_MONTH=1
   f. DAY_OF_YEAR=91
   g. DAY_OF_WEEK=5 (Sunday is 1)
   h. DAY_OF_WEEK_IN_MONTH=1 (first Thursday in this April)
   i. HOUR=11
   j. HOUR_OF_DAY=11 (24 hour clock)
   k. MINUTE=49
   l. SECOND=4

2. The Calendar class supports internationalization, using java.util.Locale.

3. The only concrete subclass of Calendar is java.util.GregorianCalendar, which provides the standard calendar. The method Calendar.getInstance() returns an instance of GregorianCalendar.

Result, P1707.java

```
java.util.GregorianCalendar[time=1364700568421,areFieldsSet=true,
areAllFieldsSet=true,lenient=true,zone=sun.util.calendar.ZoneInfo
[id="America/New_York",offset=-18000000,dstSavings=3600000,useDay
light=true,transitions=235,lastRule=java.util.SimpleTimeZone[id=A
merica/New_York,offset=-18000000,dstSavings=3600000,useDaylight=t
rue,startYear=0,startMode=3,startMonth=2,startDay=8,startDayOfWee
k=1,startTime=7200000,startTimeMode=0,endMode=3,endMonth=10,endDa
y=1,endDayOfWeek=1,endTime=7200000,endTimeMode=0]],firstDayOfWeek
=1,minimalDaysInFirstWeek=1,ERA=1,YEAR=2013,MONTH=2,WEEK_OF_YEAR=
13,WEEK_OF_MONTH=5,DAY_OF_MONTH=30,DAY_OF_YEAR=89,DAY_OF_WEEK=7,D
AY_OF_WEEK_IN_MONTH=5,AM_PM=1,HOUR=11,HOUR_OF_DAY=23,MINUTE=29,SE
COND=28,MILLISECOND=421,ZONE_OFFSET=-18000000,DST_OFFSET=3600000]

2. today=Sat Mar 30 23:29:28 EDT 2013
3. add 31 days=Tue Apr 30 23:29:28 EDT 2013
4. set date=Sat Aug 12 10:00:28 EDT 2000
5. Monday and Friday
6. one-week reservation
```

Calendar, SimpleDateFormat, DATE ARITHMETIC, EXAMPLE

P1707.java

```
import java.util.Calendar;
import java.util.Date;
import java.util.Locale;
import java.text.DateFormat;
import java.text.SimpleDateFormat;

public class P1707 {
    public static void main (String[] args) throws Exception{

/*1*/   Calendar c = Calendar.getInstance();
        System.out.println (c + "\n");

        System.out.println ("2. today=" + c.getTime());
        c.add (Calendar.DAY_OF_MONTH, 31);
        System.out.println ("3. add 31 days=" + c.getTime());
        c.set (2000, 7, 12, 10, 00);
        System.out.println ("4. set date=" + c.getTime());


/*2*/   DateFormat df =
            DateFormat.getDateInstance (DateFormat.SHORT);
        Date start = df.parse ("2/25/13");    //should be Mon
        Date end   = df.parse ("3/1/13");     //should be Fri

        SimpleDateFormat dow =             //E is day of week
            new SimpleDateFormat("E",Locale.US);
        String dowStart = dow.format (start);
        String dowEnd   = dow.format (end);
        if (dowStart.equals("Mon") && dowEnd.equals("Fri")) {
            System.out.println ("5. Monday and Friday");
        }

        c.setTime (start);
        int julianStart  = c.get(Calendar.DAY_OF_YEAR);
        c.setTime (end);
        int julianEnd  = c.get(Calendar.DAY_OF_YEAR);
        if ((julianStart+4) == julianEnd) {
            System.out.println ("6. one-week reservation");
        }
    }
}
```

EDITING NUMBERS: NumberFormat, Locale

1. Editing a number, including a percentage or currency amount, is done by combining a format object with the number.

2. Many format objects can exist in one program, and each format object can be used and/or modified multiple times.

3. Format objects are created by java.text.NumberFormat, which is abstract, and java.text.DecimalFormat which is a concrete subclass.

4. Format objects can be tailored to your needs in regard to:

   a. Number of integer and/or fractional digits
   b. Use of a grouping character, such as the comma in 12,345.
   c. International locale, meaning currency symbol and the characters that represent the decimal point and grouping.

5. NumberFormat is an abstract class. To create a format, you must call the static method for the type of format you want:

   a. NumberFormat a = NumberFormat.getInstance ();
   b. NumberFormat b = NumberFormat.getNumberInstance ();
   c. NumberFormat c = NumberFormat.getCurrencyInstance ();
   d. NumberFormat d = NumberFormat.getPercentInstance ();

6. The method getInstance returns the default number format for the current default locale. Depending on the locale, the format will be the same as the format returned by getNumberInstance, getCurrencyInstance, or getPercentInstance.

7. To get a format for a specific locale, specify a Locale as shown on line 16 on page 17.09. The Locale class is in the java.util package.

EDITING NUMBERS EXAMPLE

P1709.java

```java
import java.text.NumberFormat;
import java.util.Locale;

public class P1709 {
    public static void main (String[] args) {
        double[] d = {     .12340,
                          1.12341,
                         12.12342,
                        123.12343,
                       1234.12344,
                      12345.12345,
                     123456.12346,
                    1234567.12347  };

        NumberFormat USA =
            NumberFormat.getCurrencyInstance (Locale.US);
        for (int i=0; i<8; i++)
            System.out.println(i + ". " + USA.format(d[i]) );
        USA.setMinimumIntegerDigits (0);
        System.out.println("\nA. " + USA.format(d[0]) );

        NumberFormat frac =
            NumberFormat.getInstance ();
        System.out.println ("B. " + frac.format(d[5]) );

        frac.setMaximumFractionDigits (4);
        frac.setMinimumFractionDigits (4);
        System.out.println ("C. " + frac.format(d[6]) );

        frac.setGroupingUsed (false);
        System.out.println ("D. " + frac.format(d[7]) );
    }
}
```

Result, P1709.java

```
0. $0.12
1. $1.12
2. $12.12
3. $123.12
4. $1,234.12
5. $12,345.12
6. $123,456.12
7. $1,234,567.12

A. $.12
B. 12,345.123
C. 123,456.1235
D. 1234567.1235
```

FIXED-LENGTH NUMBERS

P1710.java

```
import java.text.NumberFormat;
import java.util.Locale;

public class P1710 {
    public static void main (String[] args) {

        NumberFormat USA =
            NumberFormat.getCurrencyInstance (Locale.US);

        String n = USA.format(1234567.89);
        System.out.println (":" + n + ":\n");

        int spacesNeeded = 16 - n.length();
        StringBuilder sb = new StringBuilder ();
        for (int i=1; i<=spacesNeeded; i++) {
            sb.append(' ');
        }
        sb.append(n);

        System.out.println (":123456789-123456:ruler line");
        System.out.println (":" + sb + ":");
    }
}
```

Result, P1710.java

```
:$1,234,567.89:

:123456789-123456:ruler line
:   $1,234,567.89:
```

1. To obtain a fixed-length string containing the number and leading spaces, prefix the formatted number with the correct number of spaces.

UNIT 18: JAVA TOOLS: jar AND javadoc

Upon completion of this unit, students should be able to:

1. Create a jar archive, display its table of contents, and extract files from it.

2. Use documentation comments in your programs so that you can use the javadoc tool to display documentation via a browser.

THE jar UTILITY

1. The jar utility is a program that can store files in a jar archive, or extract one or more files from a jar archive.

    a. The name "jar" stands for "java archive."

    b. A jar archive contains one or more other files in a compressed format.

    c. The src (source code) directory tree containing the Java API is stored in a jar archive. In recent Java versions the name of the file is src.zip rather than src.jar. The internal format of jar and zip files is the same, and files can be extracted via zip, or the filename cam be changed to src.jar and the files can be extracted via jar.

2. The files stored in a jar archive typically consist of java classes (source code or bytecode), as well as resources used by the java classes such as sound or image files.

3. Java Beans are required to be stored in jar archives, and to be identified by a jar manifest. Manifests are not covered in this unit.

4. Java classes to read and write jar archives are in the package java.util.zip.

5. jar may be used on a commandline in UNIX or in a Command Prompt DOS window. The format of a jar commandline is:

```
$  jar  options  filename(s)
```

6. Some jar options are:

    c   Create new archive
    C   Change directories during execution of jar
    f   First name in filename list is the archive file to be created or accessed
    u   Update an existing jar archive
    v   Display verbose output as jar performs its work
    t   Display the table of contents of the archive
    0   (zero) Do not use compression
    x   Extract "files" from the archive
        1) If only one filename is specified, it is the archive filename, and all "files" in it are extracted
        2) If multiple filenames are specified, the first one is the archive, and the others are specific "files" to be extracted

HOW TO USE jar

1. A jar archive's name must have the .jar filename extension.

2. To store all .class files in the current directory in a jar archive called myarchive.jar

```
$  jar  cf  myarchive.jar  *.class
```

3. To store all files in all directories in the tree below sub/topDir in a jar archive called my.jar

```
$  jar  cf  my.jar  sub/topDir
```

4. To display the table of contents of myarchive.jar

```
$  jar  tf  myarchive.jar
```

5. To extract all files from a jar archive called project.jar and place them in or below the current directory (the directory tree that was archived will be recreated)

```
$  jar  xf  project.jar
```

6. To add the file new.class to a jar archive called c.jar

```
$  jar  uf  c.jar  new.class
```

7. To add all .class files in the directory tree below subdir to a jar archive called c.jar

```
$  jar  uf  c.jar  -C  subdir  *.class
```

8. To extract the file String.java from a jar archive called src.jar and place it below the current directory in a directory called src/java/lang (the directories src, java, and lang will be created if they do not already exist). In a Windows system, use Wordpad to view the String source code.

```
$  jar  xf  src.jar  src/java/lang/String.java
```

9. To extract the file String.java from a zip file called src.zip and place it below the current directory in a directory called java/lang (the directories java and lang will be created if they do not already exist). In a Windows system, use Wordpad to view the String source code.

```
$  jar  xf  src.zip  java/lang/String.java
```

javadoc

1. The use of javadoc comments eliminates the problem of separate internal and external documentation.

2. Documentation of Java classes, methods, and variables should be embedded within source files in documentation comments.

3. The documentation comment for each class, method, or variable must immediately precede the item that it is documenting.

4. The javadoc utility is a program that extracts information from the documentation comments in source files, and uses it to create a linked set of HTML files. A separate HTML file is created for each class. An index and hierarchy tree are also created. These HTML files may be viewed via a browser.

5. The indexes and hyperlinks that javadoc creates are ONLY for the .java files that you specify on the commandline when you execute javadoc.

6. Only javadoc comments for public classes, and public and protected members, are used by javadoc, unless you specify the -private flag to request inclusion of documentation for private classes, methods, and variables.

7. Constructors are documented like methods.

8. Information about javadoc can be found in online tutorials. Search on: oracle tutorial javadoc

9. To execute the javadoc documentation generator from a commandline:

```
a.  $ javadoc AJ1009.java TCourse.java               ---UNIX
b.  C:\myjava> javadoc AJ1009.java TCourse.java      ---DOS
```

10. Two ways to view the documentation in a browser:

    a. In Windows Explorer, open your destination folder and click on the file index.html. Your default web browser will open and display your documentation.

    b. In Internet Explorer, in the entry area for web addresses, type the full pathname of your index.html. For example: C:\tahEclipse\CaseStudy\doc\index.html

    c. Internet Explorer is preferred because not all embedded javadoc tags work in Firefox or Safari.

    d. You may have to click Frames. Then click on your desired package or class.

javadoc EXAMPLE

P1805.java

```
/** The <code> P1805 </code> class contains javadoc comments.
*   @author   Teresa Alice Hommel
*   @since    3/31/11
*/
public class P1805 {

    public static final TCourse tc;

    /** The <code>main</code> method instantiates the object
    *   for the static final reference. The data is printed.
    *   @param    args   Commandline arguments in a String[]
    */
    public static void main (String[] args) {
        tc = new TCourse ("Java");
        System.out.println ("course name=" + tc.getName() );
    }
}
```

TCourse.java

```
/** The <code> TCourse </code> class continues the demo
*   of javadoc comments. Note, only the first sentence goes
*   in a method or field Summary. All sentences go in Detail.
*   @version  1.0
*/
public class TCourse {

    /** Holds training course name. It is static and final.*/
    public static final String name;

    /** Constructor initializes the String. A value must
    *   be passed because there is only one constructor.
    *   @param    name  A String to be passed to ctor.
    */
    public TCourse (String name) {
        this.name = name;
    }

    /** Typical get method. The String name is returned.
    *   @param    none
    *   @return   String
    */
    public String getName() {
        return name;
    }
}
```

```
javadoc P1805.java TCourse.java     ---run javadoc on commandline

                                  ---use mouse to click on index.html
```

HTML AND javadoc TAGS

1. Documentation comments may contain HTML. HTML headers should not be used. Commonly used HTML:

   a. <code> text </code>   The text will display in a font suitable for programming code (monospaced, or Currier).
   b. <b> text </b>   The text will display in bold.
   c. <i> text </i>   The text will display in italic.
   d. <p>   Causes a line break (end of a paragraph), and causes a blank line to be created.
   e. <br>   Causes a line break. Text that follows will start on the next line.
   f. <hr>   Causes a horizontal line across the page.
   g. <blockquote> text </blockquote>   The text will display as a separate indented paragraph.
   h. <pre> text </pre>   The text is preformated and will display as is, with indentation and line breaks, rather than being wrapped into paragraph form.
   i. <a href="url"> text </a>   The text will work as a link to the specified url.

2. Documentation comments may contain javadoc tags. Below are some tags, and the language element they are used for:

javadoc tag	used for	purpose is to identify:
@author text	classes	Author of class. May need to use javadoc -author
@deprecated text	classes methods variables	Deprecation, alternate approach. Included in .class file to get compile warnings if item is used
@exception E text	methods	Exception thrown by a method. E must be the fully qualified name of the Exception class
@param name text	methods	Name of a parameter to a method
@return text	methods	Value returned by a method
@see class @see method	classes methods variables	Hyperlink to other documentation. The link for a class can be relative or fully-qualified. The link for a method must be in the form ClassName#methodName
@since text	classes methods variables	Specific release when item was created
@version text	classes	Specific version of a class. May need to use javadoc –version

UNIT D: USING THE COMMAND PROMPT DOS WINDOW

NOTEPAD AND THE COMMAND PROMPT DOS WINDOW

1. In Windows, start a Command Prompt DOS window by clicking:

```
Start, All Programs, Accessories, Command Prompt
```

2. To change the font: right-click in the title bar, click Properties, Lucida Console, Bond Fonts, 20.

3. Do not maximize the DOS window. If you maximized it already, shrink it by pressing ALT and ENTER at the same time.

4. In DOS, change to the C drive and then change directory to the top directory of the C drive, and make a subdirectory called myjava, and change directory to myjava by typing:

```
C:               <-go to C drive if you are not in it
cd  C:\          <-go to top directory \ in the C:
mkdir  myjava    <-make a new directory called myjava
cd  myjava       <-go to your new directory myjava
```

5. In DOS, start a Notepad session to create your first program by typing:

```
notepad  MyClass.java
```

6. Notepad will ask "Do you want to create a new file?" Click:

```
Yes
```

7. Move your DOS and Notepad windows on your screen so both are visible and you can switch between them with one click.

8. In Notepad, type in your Java source program:

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

9. In Notepad, save your Java source program by clicking:

```
File
Save
```

10. Activate your DOS window by clicking anywhere in it. Then confirm that the file containing your source program is in your directory and is called MyClass.java by typing:

```
dir
```

11. The command name of the compiler is javac. In DOS, compile your source program by typing:

```
javac  MyClass.java
```

12. IF YOU GET THE ERROR MESSAGE 'javac is not recognized as an internal or external command...' it means your DOS window does not know which directory contains the java compiler. If this happens, follow these steps:

    a. Find which directory contains the java compiler, such as

    ```
    c:\Program Files (x86)\Java\jdk1.8.0_131\bin
    ```

    b. Modify the DOS path variable to include that directory by typing in a line with that same directory name, such as:

    ```
    set path=c:\Program Files (x86)\Java\jdk1.8.0_131\bin;%path%
    ```

    c. Try to compile again.

13. If there are compile errors, activate Notepad and correct the mistakes. Remember to save the revised program by clicking File, Save. Then compile again as shown in step 11

14. In DOS, after the program compiles without errors, your bytecode will be in your myjava directory in a file named MyClass.class (the name of your public class with the .class filename extension). Confirm that you have it by typing:

```
dir
```

15. The command name of the JVM is java. In DOS, execute the JVM with the name of your bytecode file WITHOUT ANY FILENAME EXTENSION by typing:

```
java  MyClass
```

16. After your program works, when you want to start a new one:

    a. Close your old Notepad.

    b. Start a new Notepad by typing in the DOS window:

    ```
    notepad  Classname.java
    ```

17. If you try to change the filename in Notepad without starting a new Notepad, when you Save As enclose your new filename in double quotes to prevent Notepad from adding the .txt filename extension.

18. REMEMBER: Java is case-sensitive. Keep your class name in the program file the same the class name in your filename.

ENVIRONMENT VARIABLES MAY HAVE TO BE SET

1. When you work in an interactive window (such as a DOS window under Windows, or with the shell in UNIX) the commandlines you type are read by a program called a command interpreter. In a DOS window your command interpreter is called the command.com, and in UNIX it is called the shell.

   Each time you enter a commandline, you are requesting to execute a program. Your command interpreter searches a small number of directories (aka folders) to locate the program. The directories to be searched are listed in a variable called PATH or path.

   a. In DOS, the names of folders listed in the path variable are separated by ; semicolons.

   b. In UNIX, the names of directories listed in the PATH or path variable are separated by : colons.

2. The java bin directory contains the executable programs that compile and execute java programs (the compiler javac and the JVM java). To locate the java bin directory, first locate the top java directory; then locate bin which is listed there.

   FOR EXAMPLE, if your java bin directory is c:\jdk1.7\bin and if your command interpreter cannot find javac or java, you can add the bin directory to your path or PATH as follows: (DO NOT USE SPACES AROUND THE = SIGN in DOS, sh or ksh)

```
a.  DOS                set  path=%path%;c:\jdk1.7\bin

b.  UNIX sh or ksh     PATH=$PATH:/jdk1.7/bin  ;  export PATH

c.  UNIX csh           setenv  PATH  ${PATH}:/jdk1.7/bin
```

3. The environment variable CLASSPATH may have to be set if javac cannot find your source file. Before setting CLASSPATH, make sure your source file is in your directory and has the correct filename. Try to compile. IF javac CAN FIND YOUR SOURCE FILE DO NOT SET CLASSPATH.

4. If javac cannot find your source file, set CLASSPATH to to contain . ("dot," which signifies the current directory). This can be done as follows:

```
a.  DOS               set CLASSPATH=%CLASSPATH%;.

b.  UNIX sh or ksh  CLASSPATH=$CLASSPATH:. ; export CLASSPATH

c.  UNIX csh         setenv CLASSPATH ${CLASSPATH}:.
```

UNIT E: ECLIPSE (based on Kepler version)

OVERVIEW, TERMINOLOGY, FOLDER STRUCTURE

1. Eclipse is a popular open-source, free IDE (Integrated Development Environment). Many IDEs such as RAD (Rational Application Developer) are based on Eclipse.

2. The Eclipse screen contains a menu bar, tool bar, and several side-by-side sections, aka visual components, called views.

   a. A view displays some resource being worked on, such as the Package Explorer or the Outline view.

   b. A perspective is a grouping of views. The default Java perspective displays the Editor in the center and other views typically used for Java application development.

3. Eclipse organizes the files and folders needed for Java application development.

   a. A workspace is the top folder for development.

```
C:\myjava\EclipseWorkspace
    Project1                  ---in Project1 the
        .settings                source code,
        bin                      bytecode, and
            com                  test files are
                training         in packages called
        doc                      com.training under
        src                      the folders src,
            com                  bin, and test. The
                training         javadocs generated
        test                     for the project
            com                  would be under the
                training         folder called doc.
    Project2
        .settings
        bin
        src
```

   b. Projects are under the workspace. One project roughly equates to one application program. Under each project:

   c. bin folder: holds your bytecode files if you choose to store source and bytecode files in separate folders.

   d. doc folder: if you generate a javadoc for your project you would typically put it in a folder called doc.

   e. test folder: if you use test drivers such as JUnit files, you would typically put them in a folder called test.

   f. .settings file: Eclipse properties, and environment variable settings.

START ECLIPSE, WELCOME SCREEN, WORKSPACE

1. Start Eclipse by clicking the icon on your desktop. If you don't have an icon on your desktop:

   a. Find the folder where Eclipse is loaded. For example: C:\Eclipse\eclipse
   b. In that folder find the icon for eclipse.exe, which is a blue sphere with four white lines across the middle.
   c. Click the eclipse.exe icon to launch it.

2. Welcome Screen: To go from the Welcome Screen to the Workbench, in the Welcome Screen click the rightmost round button with the arching silver and gold arrow. If your mouse hovers over the button, the label is "Workbench".

   a. To go from the Workbench to the Welcome screen, click Help, Welcome.
   b. Close Welcome screen: click the small X on the Welcome tab on the screen top left.
   c. The Welcome Screen has an icon for a helpful tutorial. To view it after leaving the Welcome Screen, click Help, Welcome, and then click on the Tutorial icon.
   d. For the Eclipse Java Development User Guide use your browser. Go to eclipse.org, scroll to the bottom, click on "Documentation", and look in the list on the left.

3. Workspace (see also E.02)

   a. When you open Eclipse, a "Workspace Launcher" pops up to ask for the folder name for your Workspace. Give a name under your myjava folder, for example C:\myjava\Eclipse

      1) If you click the checkbox labeled "Use this as the default and do not ask again", then the Workspace Launcher won't popup again, and all your work for this course will be under the same workspace.

      2) For this class it doesn't matter where you put your workspace, except if you know where it is, you can visit these folders via other software, such as Windows Explorer, etc.

      3) At work, unrelated projects would have separate workspaces. Their locations would be determined by project specifications.

   b. Switch between multiple workspaces: Click File, Switch Workspace.

   c. Display name of current workspace: Click File, Switch Workspace, Other. The popup displays the full path of your current workspace.

JAVA PERSPECTIVE, RESET PERSPECTIVE, VIEWS, PACKAGE EXPLORER

1. Java Perspective

   a. "Perspective" is Eclipse's term for the layout of its screen with the views for doing a specific kind of work.

   b. The Java perspective is for Java development. Eclipse has perspectives for many kinds of work: XML, Java EE, etc.

   c. Make sure you are in the Java Perspective: the Eclipse title bar, top left, should say "Java - Eclipse". If not, click Window, Open Perspective. Select "Java (default)".

   d. Restore Java perspective: click Window, Close All Perspectives. Then click Window, Open Perspective, Other. Double-click "Java (default)".

   e. You should close the Task List view in your Java perspective because it will not be used in this course.

2. Reset perspective

   a. Return to default: Click Window, Reset Perspective, Yes.
   b. Restore one view: Click Window, Show View

3. Views

   a. A view is a visual component that displays a resource being worked on, such as Package Explorer. (The Editor is not a view and does not have a tab. If you close the Editor, to open it click Window, Reset Perspective, Yes)

   b. Resize a view: Hover your mouse over the view's border until the cursor changes to a double-headed arrow. Then drag and drop the border to make the view larger or smaller.

   c. Restore a view: Click Window, Show View. Then click on the desired view.

   d. Move a view to a different location in the perspective: Press down the left mouse button over the tab of the view, drag and drop the view. You can stack views on top of each other, and then click the tab of the one you want to display at a particular time.

4. The Package Explorer view on the left side of the screen shows you the folder structure of your project(s).

   a. (default package) in the Package Explorer: this entry is Eclipse's equivalent of the UNIX or Command Prompt window's entry for "." for the current directory.

PROJECTS: CREATE NEW, RENAME, COPY, CLEAN COMPILE ALL BYTECODE

1. Create new project

   a. Click File, New, Java Project,

   b. In the "New Java Project" popup, fill in the project name. Eclipse will make the folder for it.

   c. If asked for Contents, choose "Create new project in workspace".

   d. Click the square checkbox for "Use default location"

   e. Do not change the JRE. The top round radio button should be selected by default, which should be labeled "Use an execution environment JRE" with selection "JavaSE-1.7".

   f. If asked for Project Layout, choose "Create separate folders for source and class files"

   g. Click Finish.

   h. The Package Explorer should now display your new project folder. Your src folder is under it. Your bin folder may be created now, or after your first compile when you have bytecode file(s).

2. Rename project

   a. Highlight the project in the Package Explorer.
   Click File, Rename. Enter the new name. Click OK.

3. Copy project into a new project

   a. In the Package Explorer, highlight the name of the project to be copied.

   b. Click Edit, Copy, Edit, Paste. In the "Copy Project" popup, for "Project name:" enter the name for your new project, which must be a new, non-existing name. Choose "Use default location" if you want your new project to be under the same workspace.

6. Clean compile all bytecode in a project

   a. Compile all classes in an application to give all bytecode files the current time as their timestamp: click Project, Clean. Click "Clean projects selected below".

   b. Select your projects to be cleaned. Click OK.

CLASSES: CREATE NEW, SAVE, CLOSE WITHOUT SAVING, MOVE

1. Create new class CAUTION: Your screen must be tall enough to display the entire "New Java Class" popup window.

    a. Click File, New, Class.

    b. For "Source folder" enter the name of your project followed by /src such as: MyProject/src

    c. For "Package:" enter the package such as com.training and Eclipse makes the folders if they don't exist.

        1) Alternate way (type less): Before starting to create your new class, click the package name in Package Explorer. The package and src folder names will be filled in for you in the "New Java Class" popup.

    d. For "Name:" enter the name of new class, such as P402

    e. For "Modifiers:" click button for public

    f. For "Superclass:" leave or replace java.lang.Object

    g. For "Which method stubs would you like to create?" click "Inherited abstract methods". If the class will be a main class click for "public static void main(String[] args)".

    h. Click: Finish

2. Save source code in a file (four alternate ways)

    a. Click the floppy disk icon
    b. Click File, Save
    c. Press CTRL-s
    d. Right-click in the Editor window, not in a statement, for a long menu of actions. Click Save. If Save is grayed out, your current source code has already been saved.

3. Close file without saving: Highlight the file's tab on top of the Editor. Click File, Close. A popup called "Save Resource" asks " 'ClassName.java' has been modified. Save changes?" Click No.

4. Display the source code of any class in the Java API: click the "Open Type" icon on the icon bar. Navigate to src.zip.

5. Move class to different folder: Highlight the file in Project Explorer. Click File, Move. In the "Move" popup click the folder where you want the file. Click OK. To undo, click Edit, Undo Move.

RUN, COPY SOURCE FILES

1. Run (compile, and execute if the compile succeeds)

    a. The run icon is a green circle with a white triangle.

    b. If you have not saved, the first time you click the run icon, the "Save and Launch" popup may ask which resources to save, and offers a checkbox for "Always save resources before launching".

        1) After you click that checkbox, Eclipse will save automatically before compiling and executing.

        2) The Editor allows you to undo via CTRL-Z after you save, compile, and find errors.

2. Run any application program in the Package Explorer even if it is not in the Editor

    a. Right-click on the main class in Package Explorer.

    b. Click Run, Run As, Java Application. The Console view appears at the bottom to display console output, such as from System.out.println.

3. Copy a source file into the same folder

    a. In the Package Explorer, highlight the name of the file to be copied.

    b. Click Edit, Copy, Edit, Paste. The "Name Conflict" popup asks "Enter a new name for 'OldName':" and the default is CopyOfOldName. You may enter a new name. Click OK. The classname in the new file will be CopyOfOldName or the new name that you entered.

4. Copy a source file into a different folder

    a. In the Package Explorer, highlight the name of the file to be copied. Click Edit, Copy.

    b. In the Package Explorer, highlight the name of the destination folder.

        1) To put the copy into the default package click the src folder, then click Edit, Paste.
        2) To put the copy into a folder that represents a package other than the default, such as com.training, expand src to reveal the folder with that name, highlight that folder, then click Edit, Paste. The package statement in the copied file will be updated to the new package name.

EDITOR: MAXIMIZE OR REDUCE EDITOR VIEW, * IN TAB, LINE NUMBERS, LEFT MARGIN BAR AND COLUMN, SCROLL BAR

1. Maximize and reduce the Editor view

    a. Maximize: Double-click on the classname in the tab.

    b. Restore smaller size: Double-click on classname again or click the two-rectangle Restore Button in the top right corner.

    c. Restore Package Explorer while the Editor is maximized: Click the two-node (overlapping rectangles) button in the gray column next to the Editor's top left corner.

2. * in classname tab means the class has not been saved

3. Line Numbers

    a. Right-click in the left margin bar. In the popup, select the option to show line numbers.

    b. The information bar at the bottom of Eclipse always shows line:column for the cursor position within the Editor.

4. Problem icons in left margin bar

    a. Icon clipboard with blue checkmark: appears on lines with the comment // TODO Auto-generated method stub
    It means that the method header and body is an auto-generated method stub

    b. Yellow light bulb: warning. For example, you have an import statement for a class that is not used.

    c. Red circle with white X: This line or the next line has an error that is underlined in red. Hover your mouse on the red circle or underlined text to view an explanation. Hover over the underlined text for a menu of fixes.

5. Light blue circle with plus or minus in left margin second column

    a. Plus means the entire javadoc comment is displayed
    b. Minus means only the first line with /** is displayed

6. Text lines too long to display invoke a scroll-bar

    a. A scroll-bar will appear at the bottom to enable you to scroll right and left to see the entire line. Limit your line length to prevent the need for scrolling.

EDITOR: OVERTYPE, SHORTCUTS, REFERENCE. AUTO ACTIVATION

7. Overtype text in the Editor: use your keyboard INSERT key.

8. Shortcuts

a.	CTRL-Z	Undo typing
b.	CTRL-Y	Redo typing
c.	CTRL-X	Cut
d.	CTRL-C	Copy
e.	CTRL-V	Paste

9. Control-Space (Control-space is "context sensitive″ which means it will not create a statement except inside a method.)

a.	sysout CTRL-space	System.out.println();
b.	syserr CTRL-space	System.err.println();

The following CTRL-space shortcuts cause a popup choice-box.

c.	if CTRL-space	double-click for if or if-else
d.	while CTRL-space	double-click for a while loop
e.	do CTRL-space	double-click for a do loop
f.	for CTRL-space	double-click for a for loop

g. partOrWholeClassName CTRL-space

All possible completions display in a popup. Double-click a classname or other choice. For a classname, the import statement is inserted above your class header if needed and not already coded.

10. Reference. Auto Activation

a. Type the name of a reference followed by a period. Pause typing. A popup box appears with all methodnames you can call. This won't work if the reference name has errors related to it.

b. Change the required pause duration: Click Window, Preferences. In the left column list, click Java's expand button (square with +) to expand Java subentries. Click Editor's expand button (square with +) to expand Editor subentries. Click Content Assist.

1) In the Auto Activation section there should be a check in the checkbox for Enable auto activation.
2) Set the "Auto Activation delay (ms)" to 0 to make the popup list of methods display immediately when you enter the reference identifier followed by dot.
3) The default Auto activation trigger for Java is dot.
4) Click Apply, OK.

LESS TYPING VIA SOURCE MENU: COMMENTS, INDENT, FORMAT YOUR CODE, GETTERS AND SETTERS, CONSTRUCTOR

1. Comments

   a. Comment out a section of code with /* */: Highlight the code. Click Source, Add Block Comment.

   b. Comment out a section of code with //: Highlight the code. Click Source, Toggle Comment.

2. Indent lines

   a. Indent: Highlight lines to be indented. Click Source, Shift Right.

   b. Unindent: Highlight lines to be unindented. Click Source, Shift Left.

3. Format your code

   a. Click Source, Format

   b. If some code is highlighted, only that part will be formatted. If no code is highlighted, the entire file will be formatted.

4. Getters and setters: After your variable declarations have been entered, place the cursor on the line before where you want the getters amd setters. Click Source, Generate Getters and Setters...

   1) Click checkboxes for variables that need get and set methods. (If you click the expand button (square with +) in front of the variable names, the method names are displayed. These names comply with the JavaBeans standard, which is the industry standard.

   2) Click OK.

5. Constructor: Place your cursor on the line before where you want the constructor. Click Source, Generate Constructor using Fields...

   a. Click checkboxes for variables to be initialized. For a null constructor Deselect All. Click OK.

   b. Current versions of Eclipse include the variables in the order they are declared. You must manually modify the assignments to call set methods.

LESS TYPING VIA SOURCE MENU: import *, toString(), hashcode(),
equals(), SURROUND WITH try catch,
do, for, if, while

6. import.* statements

    a. The current style is to code a separate import statement specifying package name and classname for each class to be imported.

    b. If you have used import with .* you can expand to a separate import statement per class: Click Source, Organize Imports.

7. toString method to override the method inherited from Object

    a. Place the Editor cursor in your class above the line where you want the method. Click Source, Generate toString()

    b. Select the variables to be included in the String, and change your insertion point if desired.

    c. Click for the drop-down menu of "Code Style:" and select the style you want. "StringBuilder/StringBuffer - chained calls" is popular because use of these classes reduces the load on garbage collection.

    c. Click OK.

8. hashCode and equals methods to override those inherited from Object

    a. Place the Editor cursor in your class above the line where you want the methods. Click Source, Generate hashCode() and equals()

    b. Select the variables to be included in the algorithms, and change your insertion point if desired. If desired, click "Generate method comments", "Use 'instanceof' to compare types", and "Use blocks in 'if' statements".

    c. Click OK.

9. SURROUND WITH try catch, do, for, if, while

    a. Highlight the code to be surrounded with a structure.

    b. Click Source, Surround With, and select the structure.

EDITOR: HIGHLIGHT IDENTIFIERS { } [ ] ( ) OR ONE LINE, OVERRIDING METHODS INDICATED BY ANNOTATION

1. Highlight all occurrences of an identifier

    a. Rest your cursor on any identifier, or click on the identifier, and all occurrences where it is used in your code will be highlighted.

2. Highlight from open to close { } or [ ] or ( )

    a. Double-click on the character-position after (to the right of) an open { or [or (

    b. The editor highlights to the matching close } or] or). In some versions of Eclipse you can highlight from the end to the beginning.

3. Highlight and copy a line in the editor

    a. While your cursor is anywhere in the line: press HOME to move the cursor to the first nonblank character of the line. Press HOME again to go to column 1. Then press shift-downArrow to highlight the entire line.

    b. CTRL-C, CTRL-V (the line overwrites itself), CTRL-V (the line copies to a new next line)

4. Overriding methods indicated by Annotation

    a. @Override is an annotation that Eclipse puts on the line above each method that overrides an inherited method.

        1) If the annotation is missing, that is your signal that the method does not override.

        2) If you manually enter the annotation but the method does not override, the Editor displays the red error icon in the left margin bar.

    b. A green up-triangle appears in the left margin bar next to an overriding method's header. Hover the mouse over the triangle to display the name of the ancestor class that defined the overridden method.

    c. An A appears in the Outline view next to an abstract class or method.

EDITOR: MOVE EDITOR TO METHOD OR CLASS VIA THE OUTLINE VIEW, FIND

1. Move the Editor to a method in the current or other class

    a. Hover the mouse on the methodname in your code in the Editor, hold down CTRL and click.

    b. In the popup, click Open Declaration.

2. Outline view, Move the Editor to an item in the Outline view

    a. The Outline view displays: package of the class currently displayed in the Editor, and names and categories of its members and constructors.

    b. Click on an item in the Outline view to move the Editor to that code.

    c. Outline view icons:

    1) Filled vs unfilled

        Methods: filled
        Variables: unfilled

    2) Icon shape shows member accessibility

Private	red square
Protected	yellow diamond
Public	green circle
Unspecified "package friendly"	blue triangle

    3) Letters

C	constructor
S	static
A	abstract
F	final

3. Bring a file into the Editor from Package Explorer (three alternate ways)

    a. Highlight the project in Package Explorer. Click File, Open File. Highlight the file to be read in. Click Open.

    b. Double-click on the filename in the Package Explorer.

    c. If the classname is in the current Editor: hover the mouse on the classname. Hold down CTRL and click.

4. Find any text: Click Edit, Find/Replace.

5. Move the cursor to another series of characters that are the same as highlighted text:

    a. Highlight the text to be found, such as an identifier.

    b. Press control-k. Alternatively, click Edit, Find Next.

CLASS IN A NEW PACKAGE, RENAME, CONSOLE VIEW, PROBLEMS VIEW

1. Put a class in a new package

    a. When you click File, New, Class, and fill in the New Class popup, if you fill in the package name then the folders will be created if they don't already exist.

2. Rename a source file (or any other identifier)

    a. Highlight the class name in the class header in the Editor, or in Package Explorer.

    b. Click Refactor, Rename. Type the new name. Press Enter.

    c. Warning: A variable name is changed throughout your class. However, the names of get and set methods that relate to the variable are NOT changed. Identifier changes should be done via refactoring so all occurrences of the identifier are changed.

3. Console view

    a. The console view becomes visible below the Editor when you run an application that creates output from System.out or System.err.

    b. You can raise or lower the console view height via your mouse by dragging and dropping the console upper border.

    c. Maximize: If the Console tab appears below the Editor, double-click the tab to maximize.

    d. Minimize: Double-click the Console tab.

    e. Close: Click on the X button.

    f. Open: Click Window, Show View, Console.

4. Problems view

    a. The Problems view shows errors and warnings related to the Editor contents after you save (this shows that saving also causes compiling.)

    b. After you fix an error, its red circle with X gets white or gray; after you save again the circle goes away.

    c. Display Problems view: Click Window, Show View, Problems.

COMMANDLINE ARGUMENTS, SIDE-BY-SIDE EDITORS, NAVIGATOR VIEW, FILE PROPERTIES

1. Pass commandline arguments to main

   a. Click Run, Run Configurations. In the Run Configurations popup click the tab "(x)=Arguments". Type your arguments in the "Program arguments" area. Click Run.

   b. Double quotes make multiple words appear to be one.

   c. Single quotes are treated as characters in the arguments.

   d. The arguments are not retained after the run.

2. Display 2 files in side-by-side Editors

   a. You can display and work with two or more classes side by side or above/below each other.

   b. Open both classes. Either double-click on one of the Editor tabs, or right click a tab and click Move. When a dark gray rectangle outlines that Editor, drag and drop it to the left. After the two Editors are side by side, you can drag and drop the left one to below the other.

3. Navigator view, Comparison to Package Explorer

   a. The Package Explorer shows src folders and shows files as Java artifacts, but does not show bin folders.

   b. To see all folders and the entire name of all files in your Eclipse projects, open the Navigator view:
      Click Window, Show view, Navigator.

4. Display file properties

   a. For the file that is displayed in the Editor: Click File, Properties.

   b. For any file handled by Eclipse: in the Package Explorer or Navigator View, highlight the filename. Click File, Properties.

JAR FILES: OVERVIEW, EXPORT

1. Overview

   a. Jar files can be attached to an email or placed on a shared drive to help developers work with the same code.

   b. Jar and zip files are internally the same. You can change a .jar filename extension on a jar file to .zip, and extract its contents via Windows zip features.

   c. Jar files do not have to be compressed, but typically they are. Jar uses the same compression algorithm as Winzip and other Windows 7 compression software.

2. Export files and/or a folder tree into a jar file

   a. Right click on any folder in the Package Explorer. In the menu popup click Export.

   b. In the "Export" popup expand Java and click on "JAR file". Click Next.

   c. In the "Jar Export" popup, under "Select the resources to export:"

      1) Place all source files in a jar file: click the checkbox for "Export Java source files and resources". Eclipse rebuilds the .class files when it imports; if exported JAR files are only to be used to import the classes again into Eclipse, you don't need to include the .class files.

      2) Place all .class files in the jar file: click "Export all output folders for checked projects".

   d. Under "Select the export destination:" fill in the filename for your jar file; it must be a full pathname with the foldername and filename with the .jar extension. For example: C:\myjava\myJars\u4.jar

   e. Click the checkbox for "Compress the contents of the JAR file" if you want compression.

   f. Click the checkbox for "Add directory entries".

   g. Decide whether you want to click the checkbox for "Overwrite existing files without warning".

   h. Click Next, Finish.

JAR FILES: VIEW CONTENTS, IMPORT, JRE System Library

3. View contents of a jar file

   a. Via a commandline in the same folder:

      1) Display the table of contents: `jar tf MyJar.jar`
      2) Extract all files: `jar xf MyJar.jar`

   b. Via zip software to extract or view files in a jar file:

      1) Change the filename extension from .jar to .zip
      2) Click on the file in Windows Explorer and use zip procedures.

   c. After extracting, the extracted files are text. You can view them with Notepad, Wordpad, Word, vim, etc.

4. Import a jar into an existing Eclipse project

   a. In Package Explorer, right click on the project's src folder, then in the menu popup click Import. Alternate way: highlight the src folder and click File, Import.

   b. In the "Import" popup subtitled "Select" expand General and click on "Archive File". Click Next.

   c. In the "Import" popup subtitled "Archive file" click on "Browse" and browse to the folder where the JAR file is. Double-click on the JAR file.

   d. The table of contents of the .jar or .zip file are displayed with all checkboxes checked. Uncheck any files you don't want, or "Select All" or "Deselect All".

   e. "Into folder:" This entry area lets you specify an Eclipse folder other than the project folder you preselected. If you import source files, append /src to this name so source files go in the src folder.

5. Package Explorer entry for "JRE System Library [JavaSE-1.7]"

   a. Expand this entry to view many jar files. rt.jar has the run time classes. Click its expand button to see its packages. Click the java.lang expand button to see the .class files in java.lang.

   b. Package names are usually all lower case, but there are exceptions to this convention. omg stands for Object Management Group, org.omg. It is common to use a company URL in reverse for package names, such as com.mycompany.

ALIGNMENT OF CURLY BRACES, EDITOR FONT AND POINT SIZE

1. <u>Alignment of Curly Braces</u>

    a. You can change the default alignment of curlies for a workspace, which will apply to all projects and source files in the workspace.

    b. While you have a source file in the Editor, the workspace profile is used whenever you click Source, Format to get your profile-specified alignment.

    c. A profile does not prevent you from manually entering curlies in any style.

    d. Click Window, Preferences, Java, Code Style, Formatter.

    e. Click New to create a new profile.

    f. Enter a profile name such as MyProfile.

    g. For "Initialize settings with the following profile:" use the entry "Eclipse [built-in]".

    h. Click New. The popup called "Profile 'MyProfile'" has nine tabs. Click the tab for Braces.

    i. Your choices for Brace positions are
        Same line
        Next line
        Next line indented
        Next line on wrap

2. <u>Editor Font and Point Size</u>

    a. To change the Editor font and point size in the Editor, click Window, Preferences.

    b. In the Preferences popup, in the panel on the left, expand the subtopics under General. Then expand the subtopics under Appearance.

    c. Click on Colors and Fonts. In the "Colors and Fonts" panel click the button Edit... on the right side to display the choices of fonts and sizes.

    d. Courier New is available in many sizes. Select your choice. Click OK, OK.

DEBUGGER, 1

1. The Debug icon looks like an insect. It is next to the Run icon. It requests a debugging run.

2. Breakpoints: A breakpoint is line where execution will pause. If you have not set any breakpoints Debug is the same as Run.

   a. Create breakpoint: Double-click in the left margin bar next to a procedural statement. This creates a green circle icon in the left margin.

   b. Remove breakpoint: Double-click on the breakpoint. Another way to do this: Right-click on the breakpoint icon, click Toggle Breakpoint.

3. Initiate debugging run: Click the Bug icon after you create at least one breakpoint.

   a. The "Confirm Perspective Switch" popup asks whether to open the Debug perspective. Click Yes.

   b. The Eclipse title bar changes to "Debug - name of your source file - Eclipse"

   c. The Editor, Console, and Outline views move to a horizontal arrangement at the bottom of the Eclipse window.

   d. The upper part of the Eclipse window contains three views: Debug, Variables, and Breakpoints.

4. The Debug view is on the top left. It displays the name of the application that is executing, its package and host computer, and the name of the JVM, javaw.exe (this JVM can run without being in a console window).

   a. The Debug view displays the method and line number of the NEXT line to be executed in your code, via a stack trace.

   b. In the Editor's left margin bar, a small arrow overlays the breakpoint circle of the NEXT line to be executed.

DEBUGGER, 2

5. The Variables view is top center.

   a. This view displays the names and values of variables created thus far in the current scope.

   b. You can change the values of variables by typing a new value into this view.

   c. A reference to an object or array will have an arrow to the left of its name. Click the arrow to see the variables in the object or the elements in the array.

   d. Letters to the left of an identifier are:

S	static
A	abstract
F	final
L	Local variable (parameters received by a method, or variables defined in the method)

   e. (id=123) next to a reference name is an Eclipse number. When multiple references point to the same object they have the same number.

   f. After a method executes, if a variable value changes as a result, the variable is highlighted in yellow.

   g. If you click on an identifier, its value is displayed in the bottom of the Variables view.

      1) For references, you get the result of the toString method on the object pointed to.

      2) If the inherited toString method from Object is used, you get the package-qualified classname, @, and the object's hashcode (one or more variables from the object are used to calculate an int value that the JVM uses to uniquely identify the object).

   h. To view the variables and values in any method on the stack, click the method name in the left part of the Variable view.

   i. In the Editor, the line to be executed next is highlighted in green. If you hover the mouse over a variable in this line, its value is displayed in a popup similar to the Variable view.

DEBUGGER, 3

6. The Breakpoints view is on the top right.

a. A split-bar near the bottom of this view may have to be dragged and dropped upward to display the "Hit count" and "Conditional".

1) "Hit count" lets you start the line-by-line display after your specified number of iterations of a loop.

2) "Conditional" lets you start line-by-line display after your specified condition is met.

7. Step icons (yellow arrows) above the Debug view are "Step into", "Step over", and "Step return".

a. "Step into" and "Step over" make execution go ahead one line. If the next code line to be executed calls a method, it will be executed but:

1) "Step into" means the debugger will display line-by-line execution of code in the method. You may not be able to step into a method from the API. (Eclipse may have the bytecode, but not the source code).

2) "Step over" means the debugger will not display execution of code in the method.

b. "Step return" makes the step-by-step debug display jump ahead to the return of the method you are in. The lines are executed but not traced.

8. Skip debugging display of the lines in a loop that executes many times:

a. Create breakpoints just before and just after the loop. When execution stops just before the loop, two ways to make the debugging display skip over the loop:

1) Click the Resume icon (yellow vertical line and green arrow pointing right) to jump ahead to the next breakpoint which is just after the loop.

2) Or, put the Editor cursor on the line to jump to, click Run, Run to Line.

DEBUGGER, 4

9. Step filters allow you to specify what code to skip in the the line-by-line debugging display.

   a. Select your filters: Click Window, Preferences, Java, Debug, Step Filtering. In the Step Filtering popup, you can select your filters. Note the * on the packages.

   b. After clicking Window, Preferences, in the top left entry area you can type "step" and Eclipse will fill in the items you have to click on. Most of what you use will be under General, Java, or Run/Debug.)

10. Modify code in the Editor and save while running in the debugger: With various limitations you can do this.

11. Caution: If you switch to the Java perspective while running the debugger, the debugger will pause but not terminate.

    a, If you have many debugger sessions in paused state, Eclipse will run slowly.

12. Terminate debugger: click the red square icon. The Debug view should display something like: <terminated, exit value: 0> C:\ ... \javaw.exe

    a. javaw is a version of the JVM java that works as a plug-in. It has no console window while it runs, but it displays a popup with messages when it has them.

javadoc

1. Generate javadoc documentation for a project, and put the generated files under a folder called docs that is at the same level as your src and bin folders:

   a. Highlight the project in Package Explorer. Click Project, Generate Javadoc.

   b. In the Generate Javadoc window, if the box labeled "Javadoc command:" is empty, click the "Configure..." button, navigate to the JDK bin folder where javadoc.exe is listed, such as C:\Program Files\Java\jdk1.7.0_71\bin and then click on javadoc.exe and click Open.

   c. For "Select types for which Javadoc will be generated:" your project should already be highlighted. Click to display the subfolders and source files, and click all the source files to be documented.

   d. For "Create Javadoc for members with visibility:" the default "public" is pre-selected. For additional members to be documented:
      1) "Private" selects all members.
      2) "Package" selects members with unspecified access ("package friendly"), and protected and public.
      3) "Protected" selects protected and public members.

   e. "Use standard doclet" is pre-selected. Keep this setting.

   f. For "Destination:" specify the full path of the folder where the javadocs should be placed. MAKE A NOTE OF WHERE YOU PUT YOUR JAVADOCS because the easiest way to view them in your browser is to navigate to that folder in Windows Explorer and click on index.html. Your javadocs can be placed in a central location on a server so they are available to the development team.

   g. Link your project javadocs to the JDK javadocs to enable Eclipse to display javadocs for the JDK API (for example, if you click on String, Eclipse can display the JDK String class javadoc):

      1) Click "Next>".

      2) In the box under "Select referenced archives and projects to which links should be generated:" you will see the jar files in the JRE library on the CLASSPATH of your project. Check rt.jar

2. To view your javadocs, see page 18.04.

JUnit, REFACTOR

1. Your JUnit test classes can be located under the src folder or under a separate folder for which a common name is test. To make a folder called test:

   a. In the Project Explorer, highlight and then right-click on the name of the project.
   b. In the popup click New, Source Folder.
   c. In the "New Source Folder" popup, for "Folder name:" enter the folder name (such as "test"). Click Finish.

2. Eclipse will treat the new folder as a container for packages and their classes. Compiled bytecde for these classes will be placed under the bin folder.

   a. The new folder will be added to the CLASSPATH. This means that under the test folder you can make packages with the same names as under the src and bin folders. The package statements are handled by Eclipse so that it does not matter if they are under different top folders such as src, bin, and test. If you were not using Eclipse, you would put commandline options on your commandline when you compile your classes and execute your application.

REFACTOR

1. Rename a class: Highlight the class name in the Editor. Click Refactor, Rename. A label will appear near the highlighted name "Enter new name, press Enter to refactor". Enter the new name. Press enter. The name will be changed everywhere that it is used in the workbench.

2. Rename a data member: Highlight the instance or static variable name in any place it is used. Click Refactor, Rename. A label will appear near the highlighted name "Enter new name, press Enter to refactor". Click on the down-triangle at the end of the label to get a menu. Select "Open Rename Dialog...". In the "Rename Field" popup, for "New name:" enter the new name for the variable. Select the checkboxes for "Update references", "Rename getter:", and "Rename setter:". Click OK.

3. Move a package and its classes to a different location: Highlight the package in Project Explorer. Click Refactor, Move. In the "Move" popup, highlight your Destination. Click OK.

4. Change a method signature (access modifier, return type, method name, parameter list, or exception list). Highlight the method header. Click Refactor, Change Method Signature... In the "Change Method Signature" Popup specify the changes. Click OK.

ECLIPSE EXERCISE AFTER UNIT 7 (based on Mars version)

1. If you are taking this course via BlackBoard Collaborate, before using Eclipse you must release the control-space keystroke combination in BlackBoard:

    a. In BlackBoard, click Edit, Preferences, HotKeys.
    b. Highlight "Take back control of application sharing".
    c. Click Modify. Change the keystroke combination to Control+Shift+Space.
    d. Click OK, Close.

2. If you downloaded the Eclipse zip file but have not extracted all the files yet, you must do that now.

    a. Using the zip download, Extract All into the folder C:\Eclipse.
    b. Use Windows Explorer to go to C:\Eclipse\eclipse. The icon for eclipse.exe is a blue sphere with three white lines across the middle. Make a shortcut on your desktop for eclipse.exe. IF YOU DRAG AND DROP THE ICON FOR eclipse.exe IT WON'T WORK.

3. Launch Eclipse: Click eclipse.exe or your shortcut for it.

4. Make your workspace.

    a. The "Workspace Launcher" popup asks you to "Choose a workspace folder to use for this session". For this class please use C:\myjava\eclipse and do not click "Use this as the default and do not ask again".
    b. If the "Workspace Launcher" does NOT come up, after you enter Eclipse you can click File, Switch Workspace, Other. Then use C:\myjava\eclipse and do not click "Use this as the default and do not ask again".

5. Welcome Screen, Enter Eclipse.

    a. If you have NOT opened Eclipse before, you will get the Welcome screen with icons for Overview, Samples, Tutorials, and What's New. To enter Eclipse click the silver and gold curved arrow in the upper right corner.

6. Java Perspective (this makes Eclipse arrange the screen and provide default code to assist in creating Java applications)

    a. The Eclipse title bar should say "Java - Eclipse".
    b. If the title bar says "Java EE", click Window, Perspective, Open Perspective, Java.

7. Create Project7

    a. Click File, New, Java Project.
    b. In the "New Java Project" popup, fill in Project7 for your project name (Eclipse will make the folder for it).
    c. Click the square checkbox for "Use default location"
    d. For "JRE" the top round radio button should be selected by default, which should be labeled "Use an execution environment JRE:".
    e. For "Project Layout" choose "Create separate folders for source and class files".
    f. Click Finish.
    g. The Package Explorer should now display your new project folder. Your src and bin folders are under it. Your bin folder will not be displayed in Package Explorer.

8. Create your business class. (CAUTION: Your screen must be tall enough to display the entire "New Java Class" popup window.)

    a. Highlight your project name in the Package Explorer. Click File, New, Class.

    b. For "Source folder" the name of your project followed by /src should already be filled in: Project7/src

    c. For "Package:" leave it blank.

    d. For "Name:" enter the name of your new class: RR72

    e. For "Modifiers:" click button for public

    f. For "Superclass:" leave java.lang.Object

    g. For "Interfaces:" leave the box blank.

    h. For "Which method stubs would you like to create?" click "Inherited abstract methods".

    i. Click: Finish

9. Line Numbers

   a. Right-click in the light-blue left margin bar. In the popup, select the option to show line numbers.
   b. While the Editor "has focus" the information bar at the bottom of Eclipse shows the cursor position line:column.

10. Type in eight data members in your business class, or copy these lines from notepad from your file RoomReservation72. To copy: go into notepad, highlight the lines to be copied, enter control c, move your cursor to the Eclipse Editor, click to get focus, and then paste the lines via control v.

```
private int reservationNumber;
private int seats;
private int numberOfDays;
private double dayRatePerSeat;
private double taxRate;

private double roomAmount;
private double taxAmount;
private double finalAmount;
```

11. Make Getters and setters.

    a. Place your cursor on the line before where you want the getters and setters. Click Source, Generate Getters and Setters...

       1) Click the checkboxes for reservationNumber, seats, numberOfDays, dayRatePerSeat, and taxRate.
       2) Click OK.

    b. Modify the set methods. For example:

       1) What Eclipse gives you:

```
this.seats = seats;
```

       2) Change the statements in each set method to validate the variable as appropriate. Use Eclipse's <u>Content Assist</u> for your switch, if, and println statements:

          a. <u>switch</u>: Type the characters <u>sw</u> and then press Control-Space. A popup will show the choices of switch structures. Double-click to select the type of switch you want.

b. if: Type the characters if and then press Control-Space. A popup will show the choices for if structures. Double-click to select the if you want.

c. System.out.println(): Type sysout and press Control-Space.

d. System.err.println(): Type syserr and press Control-Space.

12. Make your null constructor.

    a. Place your cursor on the line before where you want the constructor. Click Source, Generate Constructor using Fields...
    b. Deselect All. Click OK.

13. Make your constructors that accept arguments.

    a. Five-argument constructor:
        1) Place your cursor on the line before where you want the constructor. Click Source, Generate Constructor using Fields...
        2) Click the checkboxes for reservationNumber, seats, numberOfDays, dayRatePerSeat, and taxRate.
        3) Modify the assignments to call your set methods. For example:

            a) What Eclipse gives you:

```
this.seats = seats;
```

            b) What you should change it to:

```
setSeats (seats);
```

    b. Four-argument constructor: Use the procedure above for the five-argument constructor, but do not click the checkbox for taxRate. You can hard-code a literal for the taxRate when you call the five-argument constructor.

14. Create your methods calculateAmounts and printOneReservation: Type in the lines, or copy them from RoomReservation72. To copy: go into notepad, highlight the lines to be copied, enter control c, move your cursor to the Eclipse Editor, click to get focus, and then paste the lines via control v.

15. Create a main class called RR72Test. Use the procedure in paragraph 8 BUT BEFORE YOU CLICK "Finish" CLICK THE CHECKBOX FOR A main METHOD. Delete the comment template in main.

16. In the main method, create two objects of type RR72. To avoid typing the whole name RR72, type RR, then press Control-Space. Double-click on RR72 from the popup box.

17. Populate one object with valid data, and one object with invalid data. Example:

```
RR72 rr323 = new RR72(130323, 12, 5, 25.00, 0.0725);
RR72 rr444 = new RR72(130444, 14, 3, 35.00);
```

18. Call the printOneReservation method of each object.

    a. Type rr323. and then press Control Space. The popop will contain the names of methods that can be called.

19. While the main class is displayed in the Editor, execute your program.

    a. The run icon is a green circle with a white triangle.

    b. If you have not saved, the first time you click the run icon, the "Save and Launch" popup asks which resources to save, and offers a checkbox for "Always save resources before launching".

        1) After you click that checkbox, Eclipse will save automatically before compiling and executing.
        2) The Editor allows you to undo via CTRL-Z after you save, compile, and find errors.

    c. Console output from System.out.println is displayed in the Console view below the Editor.

20. Rename your class RR72 to RoomReservation72: highlight the name in the Editor in either class RR72 or RR72Test. Right-click on the name. Click Refactor, Rename... Type the new name and press ENTER.

21. Rename your class RR72Test to E72: highlight the name in the Editor in the class RR72Test. Right-click on the name. Click Refactor, Rename... Type the new name and press ENTER.

YOU DID IT! Now, you will want to know more about Eclipse!

22. To see how Eclipse identifies different errors, do the debugging exercises in Unit 2, Unit 3, and Unit 4 in Eclipse.

23. The Outline view on the right side of the Eclipse screen shows you the members of the class displayed in the Editor.

24. You may close the Task view by clicking on the X button on its tab.

25. The Package Explorer view on the left side of the Eclipse screen shows you the folder structure of your project(s). Click the expand button next to src to see your file and folders under src.

26. Eclipse has a helpful tutorial. To view it, click Help, Welcome, and then click on the Tutorial icon.

27. Eclipse has a helpful Java Development User Guide. To view it, go to eclipse.org and scroll to the bottom of the page, click on "Documentation" and look at the list on the left side of the screen.