Introduction to Java Part 2

Student Handbook

Themis

Leaders in IT Education

Advanced Topics in Java Programming

October 20, 2017

Overall Objectives

Upon successful completion of this course you will be able to use the Java language, and the classes and packages of the Java 1.8 API (Application Programming Interface) to:

1. Read, modify and create application programs that use:

    Dates, calendars, and formated numbers

    Exceptions

    Input and output operations with files

    Java Database Connectivity JDBC

    Java features for type safety: autoboxing, varargs, enumerations, assertions, and annotations

    Classes and interfaces of the Collections Framework

    Generics

    JavaBeans

2. Use JUnit to perform unit tests on classes and their methods.

3. Create javadoc documentation for your packages and classes.

4. Create jar archive files and extract files and directories from them.

This course is for programmers who have a fundamental knowledge of Java basics. For the most part this course is independent of the development environment being used for training. The only requirement is access to the Java Development Kit. However, the course is best taught using an Integrated Development Environment (IDE) such as Eclipse.

This course does not cover Java Enterprise Edition (Java EE) multi-tier enterprise applications: Servlets, Java Server Pages (JSP), Model-View-Controller (MVC) architecture, the Java EE Server environment, Apache Tomcat or other servlet containers such as WebSphere, Enterprise Java Beans (EJB), or Java web services.

October 20, 2017

Case Study Summary

Unit	Task	Copy and modify	Create
4	Create main class and business class.		CaseStudy4 RoomReservation4
5	Make an array in main, validate reservationNum, NumberFormat, StringBuilder.	CaseStudy4 RoomReservation4	CaseStudy5 RoomReservation5
6	Create subclass, helper class, 2 interfaces.	CaseStudy5 RoomReservation5	CaseStudy6 RoomReservation6 RoomResWithFood6
7	Validate dates.		FoodVendor6
8	Throw BadDataException for 4 input variables.	CaseStudy5 RoomReservation5	CaseStudy8 RoomReservation8
9	Create text file with reservation records. Read file, make array.	CaseStudy5 use RoomReservation5 as is	CaseStudy9
10	Create database rows from array elements.	CaseStudy9 use RoomReservation5 as is	CaseStudy10
12	Use autoboxing, enum, varargs, assertions, and annotation.	CaseStudy5 RoomReservation5	CaseStudy12 RoomReservation12
13	Create toString, equals and hashCode methods	CaseStudy6 RoomReservation6 RoomResWithFood6	CaseStudy13 RoomReservation13 RoomResWithFood13
14	Use ArrayList, Iterator, Collection, LinkedList, Vector, RoomReservation5	StarterCode141 CaseStudy141 CaseStudy141	CaseStudy141 CaseStudy142 CaseStudy143
15	Use HashMap, ArrayList Set, RoomReservation5	CaseStudy141 CaseStudy151	CaseStudy151 CaseStudy152
16	Use generics	E161StarterCode CaseStudy151 CaseStudy162	E161 CaseStudy162 CaseStudy163
17	Test methods using JUnit.		
18	Create javadoc documentation.		
19	Compare and sort JavaBeans.		
20	Java 1.8 new features.		

Overall Table of Contents

UNIT 1: OVERVIEW OF JAVA, Hello.java

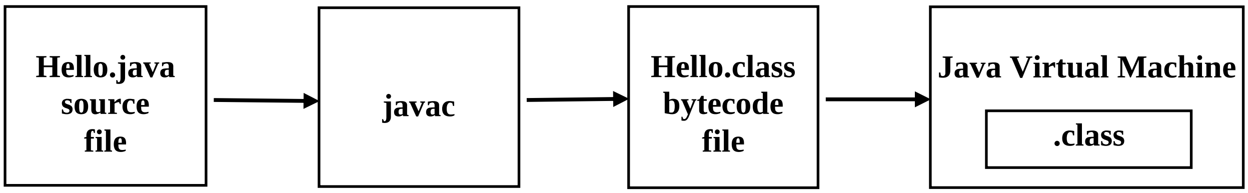
Upon completion of this unit, students should be able to:

1. Describe how to create, compile, and execute a stand-alone Java application program.

2. Describe three types of Java comments.

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

CREATE AN APPLICATION PROGRAM



1. Regardless whether you create your source code using an ASCII text editor such as notepad or vi, or an IDE such as Eclipse or IntelliJ, the source code must be compiled by javac and executed within the Java Virtual Machine, aka JVM.

2. A class is the smallest compilable unit of code. All methods and Java language statements must be inside a class. (The package and import statements are "compiler directives," and not Java language statements for the purposes of this rule.)

3. Typically, a source file contains only one class, and that class is public. If the main class is in a file with other classes, the main class must be the public class.

4. Filename requirements for source files:

    a. Only one <u>public</u> class can be in a file. The base filename must be the same as the name of the public class.
    b. The .java filename extension is required.
    c. All Java is case sensitive including filenames.

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

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

7. Compile and execute:

    a. UNIX commandline:

    ```
    $ javac Hello.java
    $ java Hello
    ```

    b. DOS commandline:

    ```
    C:\myjava> javac Hello.java
    C:\myjava> java Hello
    ```

    c. Eclipse, when the main class is in focus in the Editor:

    Click the run icon.

Hello.java APPLICATION PROGRAM, COMMENTS

Hello.java
```
1   /**
2   * Documentation comment for the class. The asterisks on line
3   * 2 and 3 are optional and will NOT be in your javadoc.
4   */
5
6   public class Hello {                            //class header
7
8       /**
9       * The documentation comment for a method must be just
10      * above the method. By default documentation is generated
11      * only for public and protected members of a class.
12      */
13      public static void main (String[] args) { //method header
14
15          System.out.print ("Hello ");
16          System.out.println ("Java\nPart 2");
17
18          //Single-line comments go from // to end of line
19          System.out.println ("Enter comments as you code.");
20
21          /*
22             Multi-line comments CANNOT BE NESTED
23             and are infrequently used.
24          */
25      }
26  }
```

Result, Hello.java
```
Hello java
Part 2
Enter comments as you code.
```

---

1. A stand-alone application must have one method called main which is where execution begins. The main method organizes the work of the application. In main's header you may see (String[] args) coded as (String... args).

2. Methods must be inside a class.

3. System.out.print and System.out.println are methods that display text on the application's console standard output.

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

5. Each simple statement must end in ; semicolon.

DOWNLOAD JDK, VIEW TUTORIALS OR JAVADOC

1. Two types of Java downloads are:

   a. JRE (Java SE Runtime Environment)

      The JRE allows end-users to run Java applications, but does not contain the compiler or other development tools.

   b. JDK (Java SE Development Kit)

      The JDK includes the JRE and also has the development tools, such as the compiler, that are needed or useful for developing applications and applets.

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" (download the version of Java that your project is using).

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".

5.

Java Release	Approximate number of classes and interfaces
Java 7	4025
Java 8	4240

UNIT 2: STATMENTS: BASIC DATA TYPES AND CONTROL STRUCTURES

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

1. Declare variables of the eight basic types with and without initial values.

2. Name the eight basic types of variables: byte, short, int, long, float, double, char, and boolean.

3. Assign the value of a literal or expression to a variable.

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

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

NUMERIC DATA TYPES, INTEGER AND FLOATING POINT

1. Numeric operations: (bitwise operations will not be covered)

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

2. Integers:

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

3. Integer literals are stored as int by default. If the letter L or l is appended, the literal is stored as a long: 123L

4. No indication is given when overflow or underflow occurs. Dividing an integer by zero causes an ArithmeticException. Dividing a float or double by zero results in a value that prints as Infinity.

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

6. Floating point:

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

7. Floating point literals are double by default. If the letter F or f is appended, the literal is stored as float: 12.3F
   E notation is accepted: 8.9E3

8. Floating-point values are stored in IEEE 754 representation to enable Java to produce the same results in all platforms.

9. Floating-point operations never throw exceptions.

10. Casting is required to assign a double to a float variable, or any floating-point value to any integer variable.

11. 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.

NUMERIC EXAMPLE

AJ203.java

```
public class AJ203 {
    public static void main (String[] args) {

/*1*/   int i = 1 + 2 * 3 - 4 / 5;
        int q = 10 / 3;                     //integer quotient
        int r = 10 % 3;                     //integer remainder
        System.out.println ("1. i="+i + ", q="+q + ", r="+r);

/*2*/   int j = 0;
        ++j;
        int k = 0;
        k++;
        System.out.println ("2. ++j=" + j + ", k++=" + k);

/*3*/   k = 4;
        i = 3 * ++k;
        System.out.println ("3. before: i=" + i + " k=" + k);

/*4*/   k = 4;
        i = 3 * k++;
        System.out.println ("4. after:  i=" + i + " k=" + k);

/*5*/   double d = 1.0 + 2.0 * 3.0 - 4.0 / 5.0;
        double dq = 9.25 / 2;               //double quotient
        double dr = 9.25 % 2;               //double remainder
        System.out.println ("5. d="+d+", dq="+dq+", dr="+dr);

/*6*/   long   varL = 2L;          //cast operator L optional
        float  varF = 1.5F;        //cast operator F required

      //i = dq;                 //possible loss of precision
        i = (int) dq;           //cast operator (int) required
        System.out.println ("6. i=" + i);
    }
}
```

Result, AJ203.java

```
1. i=7, q=3, r=1
2. ++j=1, k++=1
3. before: i=15 k=5
4. after:  i=12 k=5
5. d=6.2, dq=4.625, dr=1.25
6. i=4
```


CHAR DATA TYPE AND UNICODE

AJ204.java
```
public class AJ204 {
    public static void main (String[] args) {

        char c1 = 'A';            //graphic symbol, letter A
        char c2 = ' ';            //space
        char c3 = '3';            //character of digit 3
        char c4 = '\u0034';       //Unicode character literal
        char c5 = '\'';           //Escape sequence literal
        System.out.println (c1+" "+c2+" "+c3+" "+c4+" "+c5);

        int  i = 109;         //decimal 109 is ascii char m
      //char c = i;           //possible loss of precision
        char c = (char) i;
        System.out.println ("i=" + i + ", c=" + c);
    }
}
```

Result, AJ204.java
```
A   3 4 '
i=109, c=m
```

===

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

2. 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.

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

4. A char literal may be specified in these ways:

    a. Graphic symbol, such as 'A'
    b. Octal value in the form '\ooo' if in the range \000-\377 such as '\101' for 'A'
    c. Unicode 2-byte hex number in the form '\uhhhh' where each h is a hex digit in the range '\u0000' through '\uffff'.

5. Java recognizes these escape sequences, aka backslash escapes:

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

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

BOOLEAN DATA TYPE

AJ206.java

```
public class AJ206 {
    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");
        }   //by default, if tests a boolean for true

        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, AJ206.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 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. Use of booleans is required by these conditional and looping constructs: 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.

OPTIONAL: CASTING BASIC TYPES

1. You can cast int literals to long with l or L: 123L

2. You can cast double literals to float with f or F: 1.23F

3. 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 is automatically widened.

    c. If the receiving variable is a floating type and the source is an integer type, the integer 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.

4. 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 is cast and assigned to an integer type, 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.

5. 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;
```


STATEMENTS, CONDITIONAL STRUCTURES: if AND switch

AJ208.java
```
public class AJ208 {
    public static void main (String[] args) {
        int i=1;
        int j=2;

/*1*/   if ( (i > 0 && i < 5) || j == 3 ) {
            System.out.println ("i between 1-4, or j is 3");
        } else {
            System.out.println ("i=" + i + ", j=" + j);
        }
/*2*/   switch (j - 1) {
            default: System.out.print ("no match, ");
            case 1:  System.out.print ("match1, ");
            case 6:  System.out.print ("match6, ");
                     break;
            case 3:
            case 33: System.out.print ("3 or 33, ");
        }
        System.out.println ("after the switch");
    }
}
```

Result, AJ208.java
```
i between 1-4, or j is 3
match1, match6, after the switch
```

==

1. Statements can be simple statements ending in ; or blocks (aka compound statements) enclosed in { }.

2.
```
if (boolean_expression)       //( ) and boolean type required
    true_statement;           //code convention is to use { }
else                          //else is optional
    false_statement;
next_statement;
```

3.
```
switch (byte_short_int_or_char_expr) { //( ) and { } required
    case 1: statement1;                //As of Java 7, expr
    case 2:                            //can use String,
    case 3: case 4: statement234;      //Enum, and wrapper
    default: stmt for no_match;        //classes Byte, Short,
}                                      //Integer & Character
next_statement;
```

4. Case values must be unique constant expressions. If no case matches and no default is coded, flow of control goes to the next_statement. If no break, continue, or return is coded, flow of control falls through and case procedures are executed in the order coded.

LOOPS: while, do

<u>AJ209.java</u>

```
public class AJ209 {
    public static void main (String[] args) {

/*1*/   int i=0;                   //initialize loop control var
        while (i < 3) {                  //test loop control var
            System.out.print (i + ", ");
            i++;                    //increment loop control var
        }
        System.out.println("after, i=" + i);

/*2*/   do {
            System.out.print (i + ", ");
            i++;
        } while (i < 3);
        System.out.println("after, i=" + i);
    }
}
```

<u>Result, AJ209.java</u>

```
0, 1, 2, after, i=3
3, after, i=4
```

===

1.
```
while (boolean_expression)    //( ) and boolean type required
    true_statement;           //code convention is to use { }
next_statement;
```

2. If the boolean_expression is true, the true_statement is executed once and then flow of control goes back to 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 never executed.

3.
```
do
    true_statement;           //code convention is to use { }
while (boolean_expression);   //( ) and boolean type required
next_statement;               //unusual ; after )
```

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.

LOOPS: for

AJ210.java

```
public class AJ210 {
    public static void main (String[] args) {

/*1*/   int i;
        for (i=0; i < 3; i++) {
            System.out.print (i + ", ");
        }
        System.out.println ("after: i=" + i);

/*2*/   for (int j=0; j < 3; ++j) {
            System.out.print (j + ", ");
        }
        System.out.println ("after: j is not defined now");
    }
}
```

Result, AJ210.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 { }
   next_statement;
   ```

2. 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.

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

4. 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)

5. Variables declared in the initialization are local to the loop.
   ```

break AND LABELED BLOCKS

AJ211.java
```
1   public class AJ211 {
2       public static void main (String[] args) {
3           int i=0;
4           int j=0;
5
6   /*1*/   while (i < 5) {
7               if (i == 3) break;  //goto line 11
8               System.out.print (i + ",  ");
9               i++;
10          }
11          System.out.println ("end 1: i=" + i + "\n");
12
13  /*2*/   OUTER:
14          for (i=0; i < 3; i++) {
15              for (j=0; j < 4; j++) {
16                  System.out.print (i + "" + j + ", ");
17                  if (i==0 && j==1) break;      //goto line 20
18                  if (i==1 && j==1) break OUTER;//goto line 22
19              }
20              System.out.println ();
21          }
22          System.out.println ("end 2: i=" + i + ", j=" + j);
23      }
24  }
```

Result, AJ211.java
```
0,  1,  2,  end 1: i=3

00, 01,
10, 11, end 2: i=1, j=1
```

==

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.

3. A label can be coded on any statement.

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

---

continue AND LABELED BLOCKS

AJ212.java
```
1   public class AJ212 {
2       public static void main (String[] args) {
3           int i=0;
4           int j=0;
5
6   /*1*/   while (i < 5) {
7               //procedure for all values of i
8               if (i == 2) {  //skip rest of loop body if i is 2
9                   i++;       //increment loop var for i is 2
10                  continue;  //goto line 6
11              }
12              //more procdure for values of i that are not 2
13              System.out.print (i + ", ");
14              i++;           //increment loop var for i not 2
15          }
16          System.out.println ("end 1: i=" + i);
17
18  /*2*/   LOOP_i: for (i=0; i<3; i++) {
19              for (j=0; j<4; j++) {
20                  if (i==0 && j==1) continue;        //goto j++
21                  if (i==1 && j==1) continue LOOP_i; //goto i++
22                  System.out.print (i + "" + j + ", ");
23              }
24              System.out.println ();
25          }
26          System.out.println ("end 2: i=" + i + ", j=" + j);
27      }
28  }
```

Result, AJ212.java
```
0, 1, 3, 4, end 1: i=5
00, 02, 03,
10, 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. The continue statement can continue an outer loop, if coded with the label of a target loop.

? : TERNARY OPERATOR

AJ213.java
```
public class AJ213 {
    public static void main (String[] args) {
        int i=1;
        int j=2;
        int k;

/*1*/   if (i==j) {            //if is a statment
            k = 10;
        } else {
            k = 20;
        }

/*2*/   k = i==j ? 10 : 20;    //The ternary operator returns
                               //one of two values, and must
/*3*/   k = (i==j ? 10 : 20); //be used within a statement.


        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, AJ213.java
```
1. k=20
2. i!=j
```

==

1. The ternary operator is a conditional operator that returns the value of one of two expressions. However, in Java, the ternary operator is only an operator, not a complete statement.

2. The ternary operator has three expressions 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 returned. If the boolean_expression is false, the false_expression is evaluated and its result is returned.

4. The true_expression and false_expression cannot be void.

OPTIONAL EXERCISES

1. Create a program called E21.java that contains three loops, while, do, and for.

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

      In the body of the while loop, use an <u>if</u> construct to determine if the number is odd or even. Add the odd numbers to an odd total, and add the even numbers to an even total. After the loop finishes, display the totals.

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

      In the body of the do loop, use a <u>switch</u> to determine when the number 15 is to be printed, and print the String "***" immediately after the 15.

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

      In the body of the for loop, use <u>continue</u> to enable you to print odd numbers as is, and print even numbers with an asterisk on both sides, such as *20*

OPTIONAL SOLUTIONS

E21.java

```
public class E21 {
    public static void main (String[] args) {
        int i=0;
        int odd=0;
        int even=0;

/*a*/   while (i < 10) {
            if ( (i%2) == 0) {
                even = even + i;
            } else {
                odd = odd + i;
            }
            System.out.print (i + ", ");
            i++;
        }
        System.out.println ("o="+odd + ",e="+even + "\n");


/*b*/   do {
            switch (i) {
                case 15: System.out.print (i + "***, ");
                         break;
                default: System.out.print (i + ", ");
            }
            ++i;
        } while (i < 20);
        System.out.println ("\n");


/*c*/   for ( ; i<30 ; i++) {

            if ( (i%2) == 0) {
                System.out.print ("*" + i + "*, ");
                continue;
            }
            System.out.print (i + ", ");
        }
        System.out.println ("\n");
    }
}
```

Result, E21.java

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, o=25,e=20

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

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


(blank)

UNIT 3: METHODS AND OVERLOADING

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, SCOPE

1. A class can contain variables, methods, and other classes. These parts of the class are called its members.

2. A method is a self-contained, named, callable piece of code. All methods must be within a class.

3. A method declaration can consist of six parts:

    a. Optional modifiers, such as public or static.

        i. public — This method can be called from any class in your program.
        ii. static — This method is associated with its class, and not with a specific object of the class.

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

    c. Identifier of the method.

    d. 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.

    e. Optional Exception clauses.

    f. Block enclosing the statements of the method.

4. <u>Scope</u> means the block or section of a block where an identifier exists and is recognized by the compiler.

5. <u>Variables declared within a method block</u> are <u>local</u> within 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.

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

7. <u>Parameters received by a method</u> are also considered <u>local</u> to the method block, but are initialized by the copy of the value passed to them from the calling method.

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

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

ARGUMENTS AND PARAMETERS, return, System.exit()

ARGUMENTS AND PARAMETERS

1. A method can receive parameters from its caller, and return a return value to its caller.

2. Terminology for the values passed to a method:

   a. Arguments (sent) and parameters (received)
   b. Actual parameters (sent) and formal parameters (received)

3. A COPY of each argument value is received in the corresponding parameter variable in the called method. Changes to values in parameter variables do NOT change the calling method's argument variables that were passed.

4. For each parameter, the parameter list in parentheses must specify its data type and its identifier for use within the called method.

5. Parentheses are required in a method header, even if no parameters are coded.

return STATEMENT, RETURN VALUE, System.exit()

6. The return statement ends execution of a called method and returns control back to its caller.

7. The value returned must be a basic or reference type that is compatible with the return type specified in the method header. That is, the value must be able to be assigned to a variable of the type specified in the header without casting.

8. If no value is returned, void should be specified in the method header. 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.

9. If main() returns, control goes back to the JVM and program execution ends. Normally main does not end with return but with System.exit() which sends an exit number to the JVM. Exit numbers should be integers between 0-255 inclusive.

THE STACK

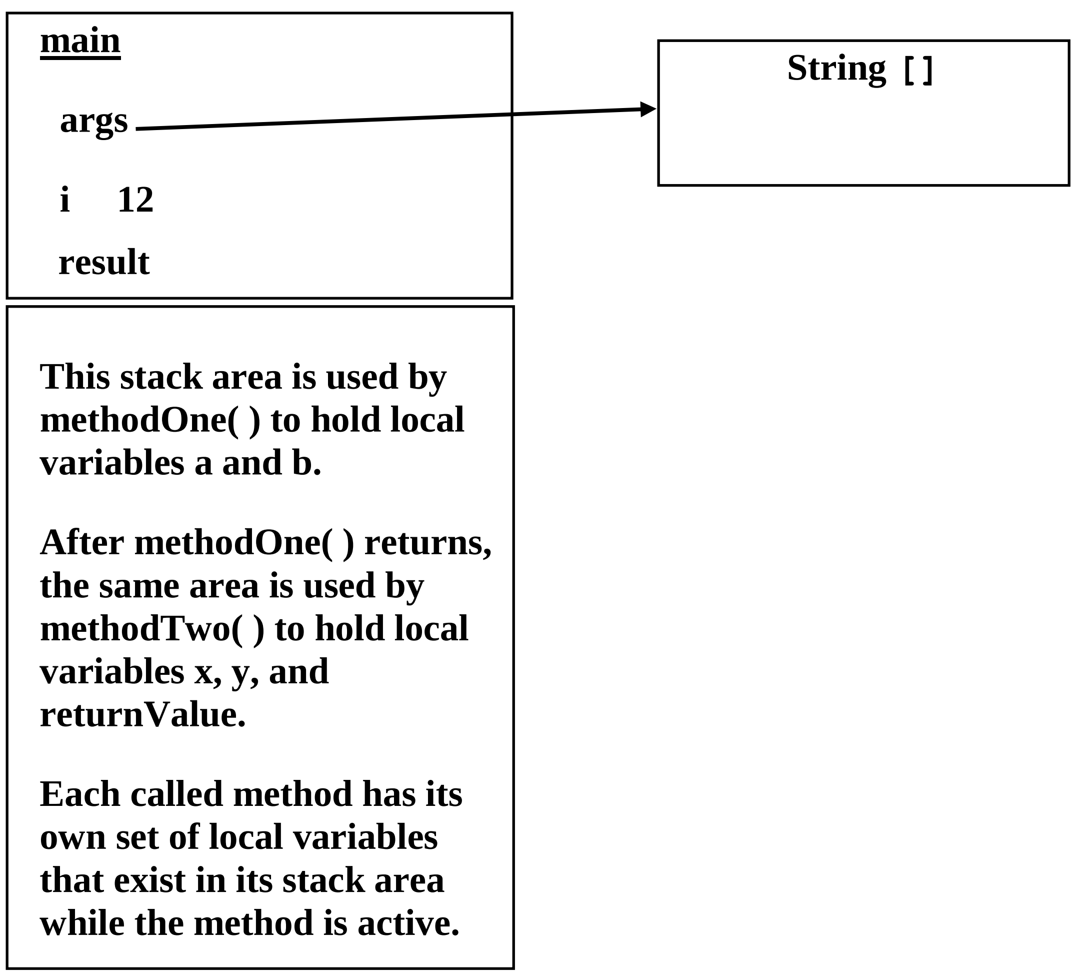
1. During program execution, for each method that is called, the JVM allocates space in a stack for the method's local variables.

   a. Local variables include (1) parameters received and (2) variables declared within the curlies of the method body.

   b. When a method is called, its stack area is allocated. When the method returns, its stack area is deallocated and that space can be used by the next method to be called.

   c. 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.

2. The stack and heap for program AJ305:



ARGUMENTS, PARAMETERS, RETURN VALUE, System.exit(), EXAMPLE

AJ305.java

```
public class AJ305 {
    public static void main (String[] args) {
        int i=1;

        methodOne (i, 2);
        System.out.println ("c. " + i);     //methodOne did
                                            //not change i

        int result = methodTwo (3, 4);      //The method call
        System.out.println ("e. " + result);//is an int expr


        //Arguments are resolved before the call to a method
        System.out.println ( "f. " + methodTwo(5, 6) );

        System.exit(0);                     //exit number
    }
    public static void methodOne (int n1, int n2) {
        System.out.println ("a. n1=" + n1 + ", n2=" + n2);
        n1 = n1 + 10;
        n2 = n2 + 10;   //2 in main is a literal, n2 is a var

        if ( (n1 + n2) < 4) {
            return;                //multiple points of return
        }

        System.out.println ("b. n1=" + n1 + ", n2=" + n2);
        return;                    //multiple points of return
    }
    public static int methodTwo (int x, int y) {
        int returnValue = 0;

        if ( (x + y) < 10) {
            returnValue = x;
        } else {
            returnValue = y;
        }
        System.out.println ("d. x=" + x + ", y=" + y);
        return returnValue;           //single point of return
    }
}
```

Result, AJ305.java

```
a. n1=1, n2=2
b. n1=11, n2=12
c. 1
d. x=3, y=4
e. 3
d. x=5, y=6
f. 6
```


METHOD OVERLOADING

AJ306.java

```
public class AJ306 {
    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 s;
        switch (day) {
            case 1: s="Monday";    break;
            case 2: s="Tuesday";   break;
            case 3: s="Wednesday"; break;  //No start days on
            default: return false;         //Thurs or Friday
        }
        boolean tmp = displayStartDay (s);
        return tmp;     //same as: return displayStartDay(s);
    }
    public static boolean displayStartDay () {
        return false;
    }
}
```

Result, AJ306.java

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

===

1. Overloading, sometimes called compile-time polymorphism, is the technique of giving two or more methods the same name and different parameter lists (the return types may be the same or different, but are usually the same).

2. The compiler calls the correct version of an overloaded method according to the arguments passed. Thus, two or more methods may NOT have the same name and parameter lists.

3. 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. Two examples are System.out.print and System.out.println.

4. One overloaded method can call another to avoid duplication of code.

OPTIONAL EXERCISES

1. Create a program called E31.java that contains an overloaded method called sum.

   a. One version of the sum method accepts two double parameters, adds them, and returns the double sum. The other version of the sum method accepts two int parameters, adds them, and returns the int sum.

   b. Call the method sum with these arguments, and display the return values on the console:

arguments	return should be
1.2 and 3.4	4.6
5 and 6	11

OPTIONAL SOLUTIONS

<u>E31.java</u>

```java
public class E31 {
    public static void main (String[] args) {

        System.out.println ("1. " + sum(1.2, 3.4) );
        System.out.println ("2. " + sum(5, 6)     );
    }

    public static double sum (double d1, double d2) {
        return (d1 + d2);
    }

    public static int sum (int i1, int i2) {
        return (i1 + i2);
    }
}
```

<u>Result, E31.java</u>

```
1. 4.6
2. 11
```

UNIT 4: CLASSES AND OBJECTS PART 1: CLASSES, OBJECTS, REFERENCES, this, STATIC AND INSTANCE MEMBERS, public AND private

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

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

2. Use the keyword this to resolve name collisions and to implement overloaded constructors.

3. Use the keyword static to differentiate between static and instance members.

4. Use the access control modifiers public and private to implement encapsulation.

5. Pass a reference argument to a method. In the called method use the received parameter to modify variables in the object.

6. Return a reference from a method.

CLASSES, OBJECTS, ENCAPSULATION, COLLABORATION

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

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.

   a. Only one class in a file can be public.

   b. The name of the public class must be used as the filename, with the .java filename extension.

4. All executable code (whether variables or methods) in a Java program must be contained in a class. Most classes contain both variables and methods.

5. To design classes, first analyze things of interest in the real-world problem domain to determine their characteristics: what <u>information</u> do we know about them, and what do they <u>do</u>. Then create classes to represent the useful characteristics.

   a. The information about things is implemented as variables, aka data attributes, data members, or properties.

   b. What things do is implemented as methods, aka method members, behavior, actions, functions, subroutines, or procedures.

6. Encapsulation is one benefit of object-oriented programming. An object encapsulates its variables and methods.

   a. <u>Encapsulation</u> means that an object is designed to be a self-contained piece of code with internal workings and data that are private so they can be changed without having to change the object's <u>public interface</u> (public methods and public static final variables) which other program statements use to work with the object.

   b. Typically, the instance variables of a class are private, and the instance methods are public.

7. <u>Collaboration</u> is the interaction of multiple objects that work together to accomplish the purposes of an application.

MORE ABOUT CLASSES AND OBJECTS

1. A class definition is Java code in a file.

2. After you define a class, you can create an object of its type. An object is an allocation of space made dynamically by the JVM during runtime in a space called the heap. An object holds one each of the non-static variables and methods defined in its class.

3. Usually when you create an object, you also create a reference variable to point to the object. When you print a reference, System.out.println and System.out.print call the toString method of the class. Classes with no toString method use the toString method of Object, which provides:

   a. The reference's class type
   b. @ at sign
   c. A hex number which is a hashcode, but which can be thought of as the symbolic address of the object.

4. The heap space allocated for an object is deallocated as soon as the object no longer has any references pointing to it.

5. Storage deallocation is performed by a process called gc, the garbage collector. gc runs at low priority. When active, it seeks and deallocates items with reference counts of zero.

6. Some object-oriented terminology:

   a. A class is sometimes called a user-defined data type.

   b. Creating an object is called "creating an instance of the class" or "instantiating" the class.

   c. The non-static variables in a class provide its "attributes" or "define the state" of an object of the class.

   d. The non-static methods in a class define the "behavior" of an object of that class.

   e. 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.

OPTIONAL: COMMANDLINE COMPILE AND RUN WITH MANY CLASSES

1. An application must have a main class, sometimes called a driver or test class, that contains the main method. To start the 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. Business classes do not contain a main method, so they cannot be executed independently as applications by themselves.

3. When an application contains many classes, each class is compiled into a separate .class bytecode file. Two ways to compile:

   a. Specify the main class only. javac will compile the source file of every class that is used by your main class and your other classes IF their source file can be found, and if their source file has been modified more recently than their most recent bytecode file was made.

      1) UNIX commandline: `$ javac M.java`
      2) DOS commandline: `C:\> javac M.java`
      3) Eclipse: When the main class is in the Editor and has focus, click the run icon.

   b. Specify the .java files that you want compiled, even if their source code files 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, which is called a "clean compile".

      1) UNIX commandline: `$ javac M.java AllOther.java`
      2) DOS commandline: `C:\> javac M.java AllOther.java`
      3) Eclipse: When the main class is in the Editor and has focus, click Project, Clean. Click "Clean projects selected below". Select your projects to be clean-compiled. Click OK.

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

   a. UNIX commandline: `$ java M`
   b. DOS commandline: `C:\> java M`
   c. Eclipse: When the main class is in the Editor and has focus, click the run icon.

new OPERATOR, CONSTRUCTORS

1. All objects are created by the new operator. There are two steps, which can be done in one statement or two.

   a. One statement:

   ```
   Course c = new Course ("UNIX", 10);
   ```

   b. Two statements:

      i. Create a reference variable to point to the object. The number in the reference will be garbage if the reference is local in a method, or all zeroes if the reference is an instance or static variable (defined outside any method).

      ii. Use new to create the object. new allocates space in the heap for the object "dynamically" (during runtime) and returns its hashcode. To complete your reference, assign the hashcode to it.

      ```
      Course c;
      c = new Course ("Java", 12);
      ```

2. The classname() used with the new operator invokes a constructor of that class. Constructors:

   a. Must have the same name as their class.
   b. Must have no specified return value.
   c. Can be called only by the new operator when you create a new object, or via this or super.
   d. Can have zero or more parameters in parentheses.
   e. Usually initialize the instance variables of their object ("initialize the internal state of the object").

3. Heap space and the static area are cleared to all bits zero before an object is created, or before a static variable is placed in the static area.

   a. Instance variables in objects contain all bits set to zero until they are set to other values.

   b. Variables with all bits zero: numbers contain 0 or 0.0, chars contain the null character '\u0000', booleans contain false, and references contain null hashcodes.

4. The JVM's class loader only loads those classes that you use, upon first mention of them while executing your code.

THE STACK AND HEAP, MEMORY ALLOCATION DURING AJ407

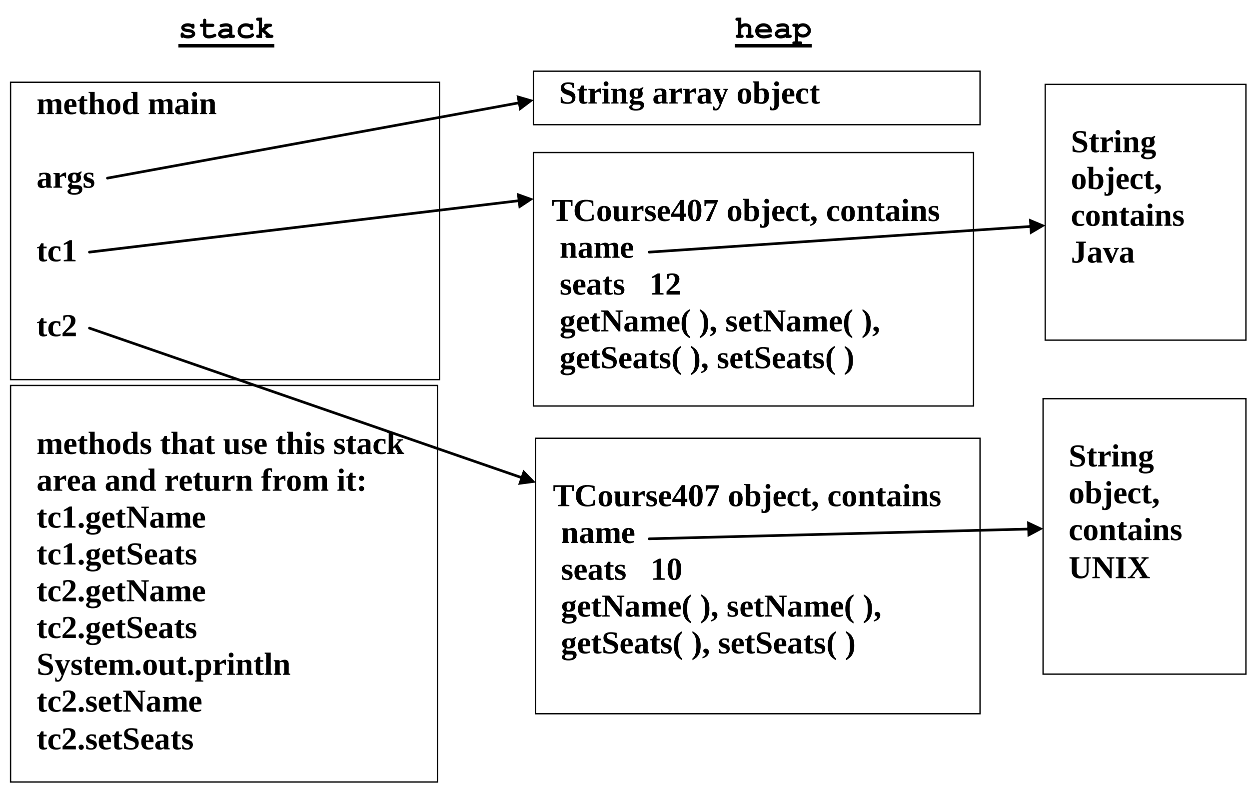
1. The JVM begins program execution by calling AJ407's main method, allocating the stack area for main, and assigning args to point to a String array that contains the commandline words that the JVM received from the execution environment.

2. The reference tc1 is created; the new operator creates a TCourse407 object in the heap, and returns the object's hashcode which is assigned to tc1. The same steps are used to create the reference tc2 and the object it points to.

3. A stack area is created for tc1.getName(); after the method 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(), etc.

4. The set methods of tc2 are called to modify the data in the object. Then more methods are called.

5. When main returns to the JVM, main's stack area is deallocated and all variables in it 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.



new AND CONSTRUCTORS, EXAMPLE

AJ407.java

```
public class AJ407 {            //main or test or driver class
    public static void main (String[] args) {
        TCourse407 tc1;
        tc1 = new TCourse407 ("Java", 12);
        TCourse407 tc2 = new TCourse407 ("UNIX", 10);

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

        tc2.setName ("Perl");
        tc2.setSeats (14);

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

        tc2 = tc1;      //2 refs to Java,12. Perl,14 is gc'ed
        System.out.println ("tc1="+tc1+", tc2="+tc2);

        tc2 = null;     //ref count of Java,12 goes down to 1
      //tc2.setName ("would cause NullPointerException");
    }
}
```

TCourse407.java

```
public class TCourse407 {                          //business class
    private String name;                           //instance vars
    private int seats;

    public TCourse407 (String newName, int newSeats) {
        setName (newName);                         //constructor
        setSeats (newSeats);
    }

    public String getName() {                      //instance
        return name;                               //methods
    }
    public void setName(String newName){
        name = newName;                            //get and
    }                                              //set methods,
    public int getSeats() {                        //aka getters
        return seats;                              //and setters
    }
    public void setSeats(int newSeats){
        seats = newSeats;
    }
}
```

Result, AJ407.java

```
Java,12, UNIX,10
Java,12, Perl,14
tc1=TCourse407@42e816, tc2=TCourse407@42e816
```


ACCESS CONTROL: public, protected, private, default

1. Access control means that each class controls whether or not other classes in the same program can directly perform these actions on instance or static members:

    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 the variable or method may be accessed by code in any other class in your program.

    b. protected means that the variable or method may be accessed by code in another class in your program if that other class is in the same package, or a subclass of this class regardless of its package.

    c. private means that the 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 can be accessed by code in any class in your program that is in the same package.

4. Access control is used to implement encapsulation. The purpose of access control is to prevent errors in the use of private and protected variables or methods by limiting access to them.

5. The public members of a class are called the "public interface" of the class.

REFERENCES

1. When you print a reference, the toString() method of its class is called to obtain a String. The default toString() method, inherited by all classes from the Object class, returns the object's class name, @ at sign, and a number displayed in hexadecimal.

    a. The hex number is a hashcode, used by the JVM as a symbolic pointer to the location in the heap where the object is located.

    b. No arithmetic can be done with the pointer value of a reference.

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

3. Every object has a 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.

4. Methods that receive a reference parameter should use an <u>if</u> to ensure that the reference is not null before dereferencing it.

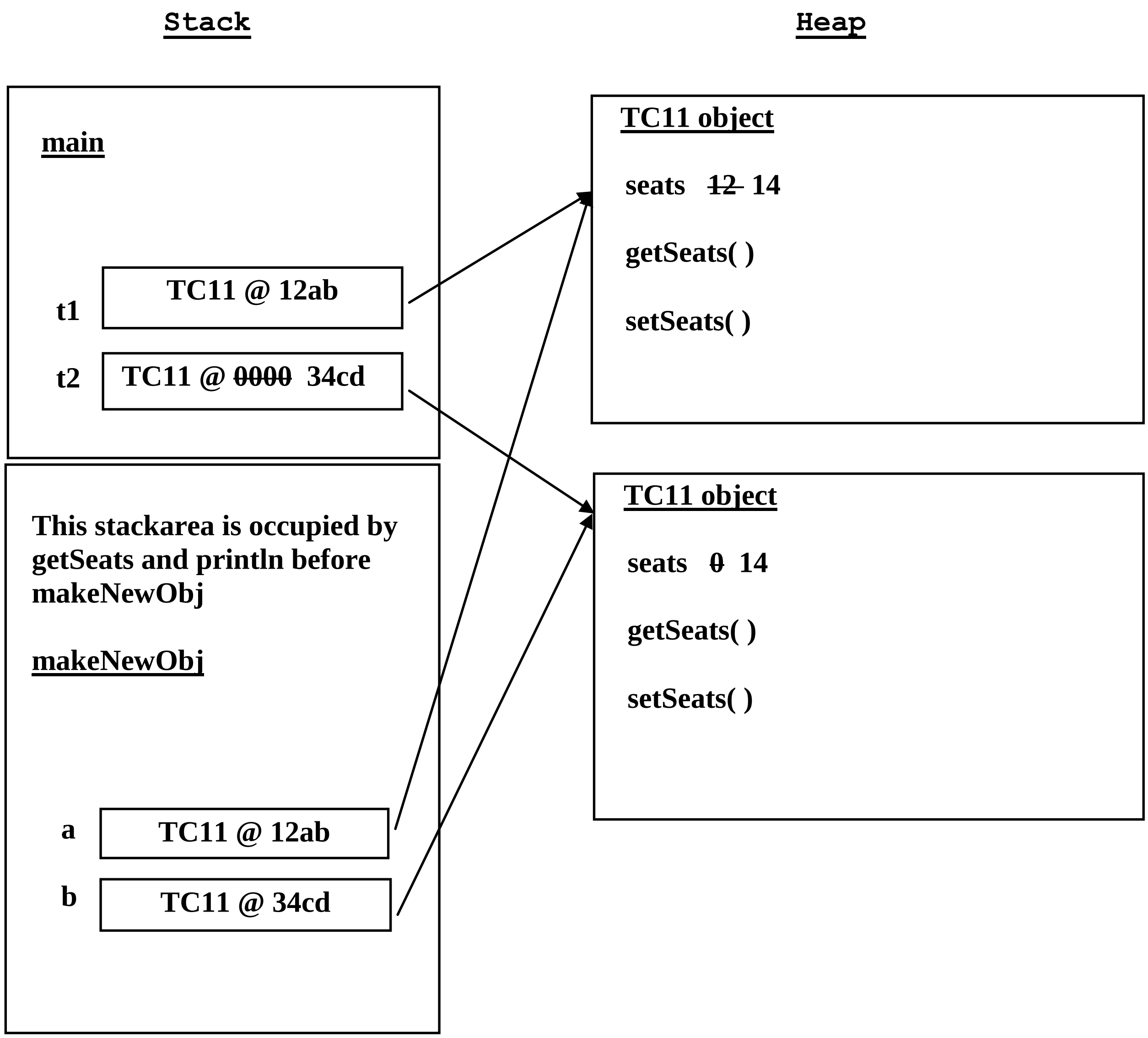
5. If two references point to objects of the same class type, one reference can be assigned to the other. Afterward both references point to the same object; the reference count of one object goes up by 1, and the other goes down by 1.

PASS A REFERENCE TO A CALLED METHOD

1. When a reference is passed to a method, the called method receives a copy of the reference. By using the copy, the called method can call accessible methods in the object.

   a. If there are accessible get methods in the object, your called method can use them to <u>obtain</u> variable values from the object.

   b. If there are accessible set methods in the object, your called method can use them to <u>change</u> variable values in the object.



PASS AND RETURN REFERENCES, EXAMPLE

<u>AJ411.java</u>

```java
public class AJ411 {
    public static void main (String[] args) {

        TC11 t1 = new TC11 (12);
        TC11 t2 = null;

        System.out.println ("1. " +
            "t1=" + t1.getSeats() + ", t2=" + t2);

        t2 = makeNewObj (t1);

        System.out.println ("3. " +
            "t1=" + t1.getSeats() + ", t2=" + t2.getSeats());
    }
    public static TC11 makeNewObj (TC11 a) {

        if (a==null) return null;

        a.setSeats (14);            //t1 now has 14 seats

        TC11 b = new TC11 (0);      //b points to new object
        b.setSeats (a.getSeats());  //b has 14 seats

        System.out.println ("2. " +
            "a=" + a.getSeats() + ", b=" + b.getSeats() );

        return b;
    }
}
```

<u>TC11.java</u>

```java
public class TC11 {
    private int seats;

    public TC11 (int newSeats) {
        setSeats (newSeats);
    }
    public int getSeats() {
        return seats;
    }
    public void setSeats(int newSeats) {
        seats = newSeats;
    }
}
```

<u>Result, AJ411.java</u>

```
1. t1=12, t2=null
2. a=14, b=14
3. t1=14, t2=14
```


this, NAME COLLISIONS, OVERLOADED CONSTRUCTORS

NAME COLLISIONS

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

2. The variable this is automatically passed to each instance method when it is called, and can be used inside the method to qualify the names of instance variables of the 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 parameter or local variable with the same identifier as an instance variable in the class, a name space collision occurs and the parameter or local variable "hides" (prevents access to) the instance variable.

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

OVERLOADED CONSTRUCTORS

4. Overloading means that a method or constructor can appear to perform similar behavior on parameters of different basic or class types.

5. Overloaded methods consist two or more methods that have the SAME NAME but DIFFERENT PARAMETER LISTS. Overloaded methods MAY HAVE THE SAME OR DIFFERENT RETURN TYPES.

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

7. 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.

8. Constructors are typically overloaded. During construction of an object one constructor can call another within the same class via the keyword this. The call to this must be the first statement in the constructor.

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

```
if (c1 != null) {
    Course c2 = new Course (c1);
}
```

this, NAME COLLISIONS, OVERLOADED CONSTRUCTORS, EXAMPLE

AJ413.java

```
public class AJ413 {
    public static void main (String[] args) {

        TC13 tc1 = new TC13 ("UNIX", 10);
        TC13 tc2 = new TC13 ("Java");
        TC13 tc3 = new TC13 ();
        TC13 tc4 = new TC13 (tc2);
           //tc4 = new TC13 (tc2.getName(),tc2.getSeats());
        TC13 tc5 = null;
        TC13 tc6 = new TC13 (tc5);   //tc5 is (TC13)null

        System.out.println (       tc1.toString() + " " +
            tc2.toString() + " " + tc3.toString() + " " +
            tc4.toString() + " " + tc6.toString() );
    }
}
```

TC13.java

```
public class TC13 {
    private String name;
    private int seats;

    public TC13 (String name, int seats) {
        setName (name);
        setSeats (seats);
    }
    public TC13 (String name) {
        this (name, -1);
    }
    public TC13 () {
        this ("none");
    }
    public TC13 (TC13 tc) { //copy ctor gets ref to same type
        this (              //avoid NullPointerException
            (tc!=null ? tc.getName()  : null),
            (tc!=null ? tc.getSeats() : -2)
        );
    }

    public String toString () {
        return "TC13:" + name + "," + seats;
    }
    public String getName() {return name;}
    public void setName(String name) {this.name = name;}
    public int  getSeats() {return seats;}
    public void setSeats(int seats) {this.seats = seats;}
}
```

Result, AJ413.java

```
TC13:UNIX,10 TC13:Java,-1 TC13:none,-1 TC13:Java,-1 TC13:null,-2
```


MODIFIER static, STATIC AND INSTANCE MEMBERS

1. A class may contain instance or static members. Static members are also called class members.

    a. Static members are defined with the keyword static.

        1) During program execution, upon first mention of any class, the JVM loads the class and scans it for static members. Exactly one of each static member is created in the static area which is in or near the heap.

        2) All identifiers in the static area are qualified by the name of their class.

        3) All objects of all classes in a program can access the accessible static variables and methods in the static area, even if no object of that class exists.

        4) Static variables are often used for public static final constants, such as Integer.MAX_VALUE.

        5) Static methods provide general functionality, such as System.arraycopy() or String.valueOf().

        6) A static method can access other static members of its own class by their simple name, but must use names qualified by a reference to an existing object to access instance members.

    b. Every object contains a set of all instance members that are defined in its class. Changing the value of an instance variable in one object does not affect the value stored in the same instance variable in another object.

        1) An instance method can access both static and instance members of its class.

        2) In an instance method, static variables or methods of the same class can be accessed as name, this.name (NOT proper style), or ClassName.name.

2. "Dot" notation is used to refer to members of a class.

    a. Names of instance members are qualified by the reference that points to their object, such as myArray.length.

    b. Names of static members are qualified by the name of the class, such as Integer.parseInt().

3. Static variables are used to specify default values, and to create total accumulators and number generators.

MODIFIER static, STATIC AND INSTANCE MEMBERS, EXAMPLE

AJ415.java

```
public class AJ415 {
    public static void main (String[] args) {

        System.out.println (
            "Number of courses scheduled: " +
            TC15.getNumScheduled() );

        TC15 tc1 = new TC15 ("Java", 12);
        TC15 tc2 = new TC15 ("UNIX", 10);

        if ( TC15.getNumScheduled() == 2) {
            System.out.println ("2 courses: " +
               tc1.toString() + "  " + tc2.toString() );
        }
    }
}
```

TC15.java

```
public class TC15 {

    public static final double SEAT_COST = 50.00; //
    private static int numScheduled = 0;          // static
    public static int getNumScheduled() {         // members
        return numScheduled;                      //
    }                                             //

    private String name;                          //
    private int seats;                            //
    private int myNumber;                         //
    private double classCost;                     // instance
                                                  // members
    public String toString () {                   //
        return "TC15:" + myNumber + "," + name    //
            + "," + seats + "," + classCost;      //
    }                                             //

    public TC15 (String name, int seats) {        //
        this.name = name;                         // ctor
        this.seats = seats;                       //
        numScheduled++;                           //
        myNumber = numScheduled;                  //
        classCost = seats * SEAT_COST;            //
    }                                             //
}
```

Result, AJ415.java

```
Number of courses scheduled: 0
2 courses: TC15:1,Java,12,600.0  TC15:2,UNIX,10,500.0
```


EXERCISES

Notes

To learn faster and remember longer, study the exercise and provided solution before creating your own solution with Eclipse.

If you don't know Eclipse, you can learn it now by following the steps in Appendix E, pages ajE.27-ajE.32, which show how to create and execute the classes for the Unit 4 exercise.

1. The case study for this course is a room reservation application for a training center. The main class is CaseStudy4.java. The business class is RoomReservation4.java.

   In Eclipse, make a new project called MyJava2 and place your source code under a package called com.themisinc.u04.

   RoomReservation4.java in package com.themisinc.u04

   a. Create three public static final constants:

type	constant name	value
int	DEFAULT_SEATS	12
int	DEFAULT_NUMBER_OF_DAYS	5
double	DEFAULT_DAY_RATE_PER_SEAT	25.00

   b. Create four variable declarations for input data, and get and set methods for each one. In the set methods, use the same identifier for the parameter as the instance variable to be set, and use this to resolve the name collisions. If the value for a variable is invalid, print an error message via System.err.println, and set the variable to the default constant.

type	variable name	valid values	structure to use
int	reservationNumber	no validation yet	
int	seats	10, 12, and 14	switch
int	numberOfDays	1 through 5	if
double	dayRatePerSeat	25.00 through 65.00	if

   c. Create a variable declaration for a calculated amount:

type	variable name
double	roomAmount

d. Create three public constructors:

   1) A null constructor that receives no parameters and contains either no statements, or one statement only: super();

   2) A constructor that receives four parameters and calls the appropriate set method for each value received. The parameters are:

      reservationNumber
      seats
      numberOfDays
      dayRatePerSeat

   3) A constructor that receives three parameters and passes them, along with DEFAULT_DAY_RATE_PER_SEAT to the four-argument constructor. The parameters are:

      reservationNumber
      seats
      numberOfDays

e. Create a private void method called calculateAmount that receives no parameters and calculates the roomAmount:

variable name	calculation
roomAmount	product of seats, numberOfDays, and dayRatePerSeat

f. Create a public void method called printOneReservation that receives no parameters, calls the method calculateAmount, and then prints the values of all the input variables and the roomAmount followed by two \n newlines to create a blank line at the end of the printout.

   Your report can have a format that is different from the provided solution. Formatting of numbers will be covered in a later unit.

CaseStudy4.java in package com.themisinc.u04

g. Create two or more RoomReservation4 objects, and call the printOneReservation method for each one.

h. Execute your program several times, changing the values passed to the constructors for each variable, to test your logic and make sure that each value is correctly validated.

SOLUTIONS

CaseStudy4.java in com.themisinc.u04

```
package com.themisinc.u04;
public class CaseStudy4 {
    public static void main (String[] args) {

        RoomReservation4 rr1 = new RoomReservation4 (
            130323, 12, 5, 25.00);
        rr1.printOneReservation();

        RoomReservation4 rr2 = new RoomReservation4 (
            130445, 14, 3);
        rr2.printOneReservation();
    }
}
```

RoomReservation4.java in com.themisinc.u04

```
package com.themisinc.u04;
public class RoomReservation4 {

    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;

    private int reservationNumber;          //4 "input" vars
    private int seats;
    private int numberOfDays;
    private double dayRatePerSeat;

    private double roomAmount;              //calculated var

    public RoomReservation4 () {        //no-arg constructor
    }

    public RoomReservation4 (           //4-arg constructor
      int reservationNumber, int seats,
      int numberOfDays, double dayRatePerSeat) {
        setReservationNumber (reservationNumber);
        setSeats (seats);
        setNumberOfDays (numberOfDays);
        setDayRatePerSeat (dayRatePerSeat);
    }

    public RoomReservation4 (           //3-arg constructor
        int reservationNumber,
        int seats,
        int numberOfDays
    ) {
        this (reservationNumber, seats, numberOfDays,
            DEFAULT_DAY_RATE_PER_SEAT);
    }

```

```
    private void calculateAmount () {
        roomAmount = seats * numberOfDays * dayRatePerSeat;
    }

    public void printOneReservation () {
        calculateAmount ();
        System.out.println (
           "Reservation:        " + reservationNumber +
         "\nNumber of seats:    " + seats +
         "\nNumber of days:     " + numberOfDays +
         "\nDay rate per seat: " + dayRatePerSeat +
         "\nRoom amount:        " + roomAmount + "\n");
    }

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

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

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

```



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

Result, CaseStudy4.java in com.themisinc.u04

```
Reservation:        130323
Number of seats:    12
Number of days:     5
Day rate per seat:  25.0
Room amount:        1500.0

Reservation:        130445
Number of seats:    14
Number of days:     3
Day rate per seat:  25.0
Room amount:        1050.0
```

Another style of main class that tests the constructor and individual methods:

RoomReservation4Test.java in com.themisinc.u04

```
package com.themisinc.u04;

public class RoomReservation4Test {
    public static void main (String[] args) {

        p ("\nTest constructor with good data\n");
        RoomReservation4 rr1 = new RoomReservation4 (
            130323, 12, 5, 25.00);

        p ("setReservationNumber, expected: 130323");
        p ("getReservationNumber, actual:   " +
            rr1.getReservationNumber() );
        p ("setSeats, expected: 12");
        p ("getSeats, actual:   " + rr1.getSeats() );
        p ("setNumberOfDays, expected: 5");
        p ("getNumberOfDays, actual:   " +
            rr1.getNumberOfDays() );
        p ("setDayRatePerSeat, expected: 25.00");
        p ("getDayRatePerSeat, actual:   " +
            rr1.getDayRatePerSeat() );

```



```
        p ("\nTest individual methods with good data\n");
        RoomReservation4 rr2 = new RoomReservation4 ();

        int reservationNumber = 130445;
        int seats = 14;
        int numberOfDays = 3;
        double dayRatePerSeat = 35.00;

        rr2.setReservationNumber (reservationNumber);
        rr2.setSeats(seats);
        rr2.setNumberOfDays(numberOfDays);
        rr2.setDayRatePerSeat(dayRatePerSeat);

        p ("setReservationNumber, expected: " +
            reservationNumber);
        p ("getReservationNumber, actual:   " +
            rr2.getReservationNumber() );
        p ("setSeats, expected: " + seats);
        p ("getSeats, actual:   " + rr2.getSeats() );
        p ("setNumberOfDays, expected: " + numberOfDays);
        p ("getNumberOfDays, actual:   " +
            rr2.getNumberOfDays() );
        p ("setDayRatePerSeat, expected: " + dayRatePerSeat);
        p ("getDayRatePerSeat, actual:   " +
            rr2.getDayRatePerSeat() );
    }
    public static void p (String s) {
        System.out.println (s);
    }
}
```

Result, RoomReservation4Test.java in com.themisinc.u04

```
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

Test individual methods with good data

setReservationNumber, expected: 130445
getReservationNumber, actual:   130445
setSeats, expected: 14
getSeats, actual:   14
setNumberOfDays, expected: 3
getNumberOfDays, actual:   3
setDayRatePerSeat, expected: 35.0
getDayRatePerSeat, actual:   35.0
```


OPTIONAL EXERCISE E42.java

Explain how the following program works, why line 4 is used, and why the methods on lines 27 and 30 must be static.

<u>E42.java</u>

```
1   public class E42 {
2       public static void main (String[] args) {
3           TC42 ref1 = new TC42 ("UNIX", 10);
4           ref1 = null;
5           TC42 ref2 = new TC42 (ref1);
6           System.out.println (ref1 + ", " + ref2);
7       }
8   }
9   class TC42 {
10      private String name = "default1";
11      private int seats = -1;
12
13      public TC42 (String name, int seats) {
14          setName (name);
15          setSeats (seats);
16      }
17      public TC42 (TC42 e) {
18          this (
19              (e!=null ? e.getName()  : TC42.getDefaultS() ),
20              (e!=null ? e.getSeats() : TC42.getDefaultI() )
21          );
22      }
23
24      public String toString () {
25          return "TC42:name=" + name + ",seats=" + seats;
26      }
27      protected static String getDefaultS() {
28          return "default2";
29      }
30      protected static int getDefaultI() {
31          return -2;
32      }
33
34      public String getName() { return name; }
35      public void setName(String name) { this.name = name; }
36      public int  getSeats() { return seats; }
37      public void setSeats(int seats) { this.seats = seats; }
38  }
```

<u>Result, E42.java</u>

```
null, TC42:name=default2,seats=-2
```

UNIT 5: ARRAYS, String, StringBuffer, StringBuilder, WRAPPER CLASSES Integer AND Character, Number Formats

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

1. Declare and work with single-dimensional array references and objects, including array elements, the array length attribute, and the method System.arraycopy().

2. Briefly describe what a String is, and how String differs from StringBuffer and StringBuilder; locate documentation for these classes and use it to work with their methods.

3. Obtain commandline arguments as an array of Strings.

4. Briefly explain the purpose of the wrapper classes, and use the methods and variables of Integer, and the methods of Character.

5. Use the NumberFormat and Locale classes to edit a number for report printing.

ARRAY DECLARATIONS, ARRAY ELEMENTS

1. Arrays are stored in the heap as objects and require the use of references. Elements that you do not initialize have all bits zero, so ints contain 0; doubles contain 0.0, etc.).

2. Arrays have "special support" in Java:

   a. Arrays 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. All elements in an array must have the same data type, which can be a basic or class type. The number of elements is specified when the array is allocated, and is final.

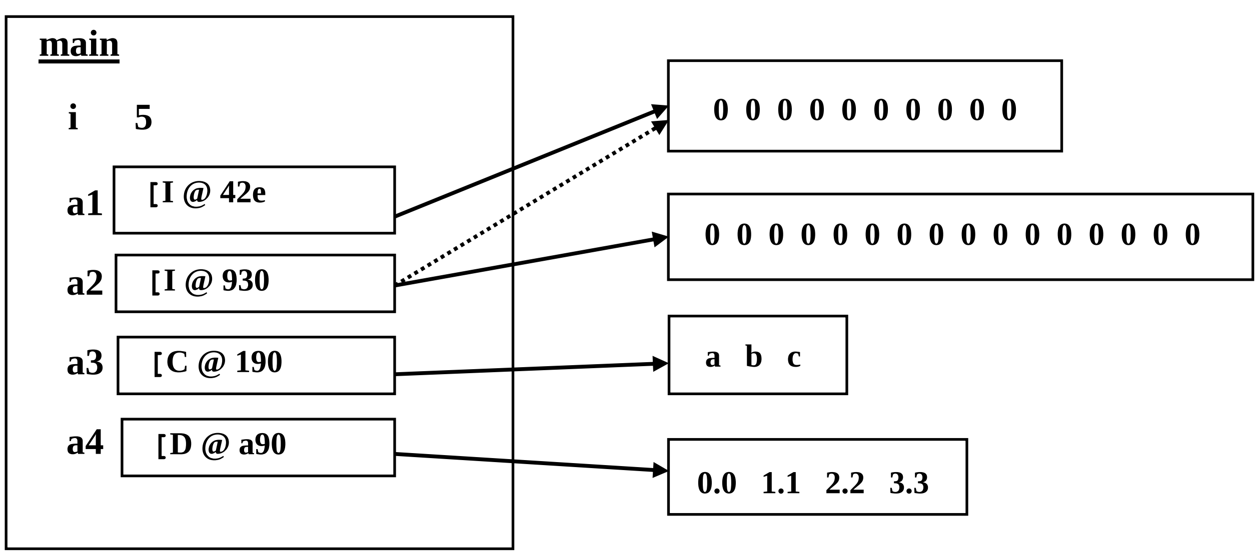
4. Array elements are used like variables to contain values, and can be used in any expression valid for their data type.

5. The identifier of an element is the array name and a subscript (aka index) enclosed in [ ] square brackets.

   a. Subscripts must be a non-negative integer. The initial element is subscript 0. The subscript of the last element is the number of elements minus 1.

   b. An ArrayIndexOutOfBoundsException occurs when a subscript is outside the range 0 through the array length minus 1.

6. Arrays are a subclass of Object and inherit from Object. A reference of type Object[] can point to an array of objects of any class type.



ARRAY DECLARATIONS AND ELEMENTS, EXAMPLE

<u>AJ503.java</u>

```
public class AJ503 {
    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'};      //reference and array

        double a4[] = {0.0, 1.1, 2.2, 3.3};    //older syntax

        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
        System.out.println ("a1=" + a1 + ", a2=" + a2);
    }
}
```

<u>Result, AJ503.java</u>

```
i=5, a1=[I@42e816, a2=[I@9304b1, a3=[C@190d11, a4=[D@a90653
a
b
c
a1=[I@42e816, a2=[I@42e816
a1=[I@42e816, a2=null
```

===

1. If two references point to the same array, either one of them can be used to access or modify it.

2. If two references point to arrays with the same type of elements (such as int, byte, 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.

3. The two notations <u>int[] a1</u> and <u>int a1[]</u> mean the same.

---

arrayname.length AND System.arraycopy()

<u>AJ504.java</u>

```
public class AJ504 {
    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] + "  ");

            for (i=0; i < a2.length; i++)
                System.out.print ("*" + a2[i] + "*  ");
            System.out.println ();
        }
    }
}
```

<u>Result, AJ504.java</u>

```
20  21  12  13  12  13  14  15  16  19  *20*  *21*
```

===

1. Each array automatically has a length variable called arrayname.length which is final. The length variable should be used in loops to prevent an ArrayIndexOutOfBoundsException which will occur if a subscript goes out of bounds.

2. A simple loop can copy element values individually from one array to another, if element types are compatible or cast.

3. The method System.arraycopy() can copy elements from one array to another, or within the same array. The values of overlapping elements are not propagated.

4. Five arguments must be passed to System.arraycopy():
    a. sourceArrayName
    b. subscriptOfFirstElementToBeCopied
    c. destinationArrayName
    d. subscriptOfFirstElementToBeOverwritten
    e. numberOfElementsToBeCopied

ARRAYS OF OBJECTS

AJ505.java

```
public class AJ505 {
    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, AJ505.java

```
0=true, 1=false, 2=true, 0=CA, 2=TX,
[Z@10948bd   [Ljava.lang.String;@8697ce
```

```
   stack                  heap
                                     ______________________________
[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                 |
                  |_________________________| |____________________|
```


String

AJ506.java

```
public class AJ506 {
    public static void main (String[] args) {

        System.out.println (50 + "" + 7 + " a\"a" + " b'b");

        String s1 = "April in Paris";
        String s2 = new String ("Christmas in Moscow");
        s2 = s1;
        System.out.println (s1 + ",  " + s2);

    }
}
```

Result, AJ506.java

```
507 a"a b'b
April in Paris,  April in Paris
```

1. A string is a sequence of zero or more characters stored in an object of type String.

2. A String literal is compiled into a String object, and an internal, compiler-created reference points to the object.

3. The \ backslash character in a String has to be coded as the escape sequence \\. Unicodes can be used in Strings to designate any character except newline and return, which must be coded as \n and \r.

4. 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.

5. A String is NOT a char array, and the following will NOT compile: char[] Str = "abc";

6. If you pass a reference to be printed by System.out.print or System.out.println, these methods call ref.toString().

7. To compare the data in two String objects use s1.equals(s2) because s1==s2 compares the references and returns true only if they refer to the same object.

String LITERAL POOL

AJ507.java

```
public class AJ507 {
    public static void main(String[] args) {

/*1*/   String s1 = new String("Hello");      //explicit new
        String s2 = new String("Hello");

        if (s1 == s2) {
            System.out.println("s1 == s2 is true");
        } else {
            System.out.println("s1 == s2 is false");
        }

/*2*/   String s3 = "wonderful";              //implicit new
        String s4 = "wonderful";

        if (s3 == s4) {
            System.out.println("s3 == s4 is true");
        } else {
            System.out.println("s3 == s4 is false");
        }

/*3*/   if ("x" == "x") {       //javac-generated references
            System.out.println("x == x is true");
        }
    }
}
```

Result, AJ507.java

```
s1 == s2 is false
s3 == s4 is true
x == x is true
```


THE HEAP FOR AJ509.java

```
                         _______________________________________
String[]args----------->|                                       |
[Ljava.lang.String;@86c |   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@cd567  | String@ef9ab  |   |
                        |   |_______________|_______________|   |
                        |       /                  \            |
                        |______/____________________\___________|
                              /                      \
                             /                        \
                            /                          \
cd567______________________/__    ef9ab_________________\_______
|                             |   |                             |
|  String object              |   |  String object              |
|                             |   |                             |
|  args[0].length() returns 11|   |  args[1].length() returns 7 |
|   _____________             |   |   _________                 |
|  |             |            |   |  |         |                |
|  | commandline |            |   |  | example |                |
|  |_____________|            |   |  |_________|                |
|                             |   |                             |
|  charAt()                   |   |  charAt()                   |
|  equals()                   |   |  equals()                   |
|  length()                   |   |  length()                   |
|_____________________________|   |_____________________________|
```


COMMANDLINE ARGUMENTS AND String[] args

AJ509.java

```
public class AJ509 {
   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, AJ509.java with 2 arguments: commandline example

```
0. commandline
1. example
strlen=11
```

===

1. When you execute your program on a commandline, the command interpreter (UNIX shell or Command Prompt cmd.exe) stores the commandline words as elements of a String array, then passes the array to the JVM. The JVM strips off the program name and passes the array to the program's main method when program execution starts. By convention the array is called args.

2. Your programming environment dictates how you can enter commandline arguments:

   a. UNIX: $ java AJ509 commandline example

   b. DOS: C:\myjava> java AJ509 commandline example

   c. Eclipse: Click Run, Run Configurations. In the popup window click the tab "(x)=Arguments". Type your arguments in the "Program arguments" area. Click Run.

3. When an array contains basic types, the basic variables are in the array object. 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.

4. A "pure java" program should follow POSIX conventions for commandline options and arguments.

OPTIONAL: HEAP FOR AJ511.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

AJ511.java

```
public class AJ511 {

   public static final int ROWS = 2; //Arrays are "row-major"
   public static final int COLS = 4; //Dimension 1 is rows
                                     //Dimension 2 is columns
   public static void main (String[] args) {

/*1*/  int[][] a1 = {{100,101,102,103},{110,111,112,113},};

       for (int rows=0; rows<ROWS; rows++) {

            for (int cols=0; cols<COLS; cols++) {
                System.out.print (a1[rows][cols] + "  ");
            }
            System.out.println ();

        }

/*2*/   int [][] a2 = new int[ROWS][COLS];

        for (int rows=0; rows<a2.length; rows++) {

            for (int cols=0; cols<a2[rows].length; cols++) {

                a2[rows][cols] = 200 + rows*10 + cols;
                System.out.print (a2[rows][cols] + "  ");
            }
            System.out.println ();

        }
    }
}
```

Result, AJ511.java

```
100  101  102  103
110  111  112  113
200  201  202  203
210  211  212  213
```

===

1. The values for array a1 can be coded indented as follows:

```
    int[][] a1 = {
        {100,101,102,103},
        {110,111,112,113},
    };
```


StringBuffer, StringBuilder

AJ512.java

```
public class AJ512 {
    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 () {
        StringBuffer sb = new StringBuffer("StringBuffer:");
        sb.append("varI=").append(varI);
        sb.append(",varD=").append(varD);
        return sb.toString();
    }

    public static String useStringBuilder () {
        return new StringBuilder("StringBuilder:")
            .append("varI=")
            .append(varI)
            .append(",varD=")
            .append(varD)
            .toString();
    }
}
```

Result, AJ512.java

```
StringBuffer:varI=123,varD=4.5
StringBuilder:varI=123,varD=4.5
```

==

1. String objects are unchangeable. Each concatenation of Strings creates a new String object, deallocates an old one, and increases the work of the garbage collector.

```
String s = "a";   //creates object with "a"
s = s + "b";      //deallocates object with "a" and
                  //creates new object with "ab"
```

2. StringBuffer or StringBuilder should be used for operations that modify strings (such as appending, concatenating, inserting, deleting) to avoid increased garbage collection. Both classes allow the same modifications of data contained in the object.

3. StringBuffer is a thread-safe class. StringBuilder, new in Java 5, is not thread-safe, which may make it more efficient.

WRAPPER CLASS Character, EXAMPLE

AJ513.java

```
public class AJ513 {
    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)) {                      //an equals method
        System.out.println ("9. true");           //tests same class
    } else {                                      //and same values
        System.out.println ("10. false");         //in "important"
    }                                             //variables

    String s = obj1.toString();
    System.out.println ("11. " + s);
    }
}
```

Result, AJ513.java

```
1. false
2. true
3. true
4. true
5. false
6. true
7. A
8. 1
10. false
11. 1
```


OVERVIEW OF THE WRAPPER CLASSES AND THE Number CLASS

1. The wrapper classes are defined in java.lang. They are:

   a. Number
   b. Byte, Short, Integer, Long, Float, and Double
   c. Character
   d. Boolean
   e. Void

2. The wrapper classes provide:

   a. A way to encapsulate the value of any basic type variable into an object.
   b. Methods to perform commonly-needed tasks.
   c. Constants that specify the maximum and minimum values that can be stored in variables of the basic types, such as Integer.MAX_VALUE and Integer.MIN_VALUE.

3. Wrapper classes enable you to

   a. Pass basic type values as object arguments to methods that require object type parameters (another approach is autoboxing).
   b. Store basic type values in classes that require their values to be stored in objects, such as classes of the Collections Framework and the Reflection API.

4. Number is the abstract superclass of wrappers for the numeric basic types: byte, short, int, long, float, and double.

5. The Number class defines the following six instance methods. All are abstract except for byteValue() and shortValue(). All numeric wrapper classes implement all six 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.

6. Because all subclasses of Number implement the above methods, the numeric value stored in any numeric wrapper class object can be retrieved as the value of any basic numeric type. However, if the data type of the object is not the same as the data type of the return value, rounding and truncation may occur.

WRAPPER CLASS Integer, EXAMPLE

AJ515.java

```
public class AJ515 {
    public static void main (String[] args) {

/*1*/   int i = 1;                    //basic type int with 1
        Integer ref = new Integer (i);//ref to Integer with 1
        p ("1. i=" + i + ", r=" + ref);

/*2*/   long n = 2L;
        if (n>=Integer.MIN_VALUE && n<=Integer.MAX_VALUE) {
            ref = new Integer ( (int)n );          //ref to 2
            p ("2. fits in Integer: " + ref);
        }

/*3*/   //Convert String with digit chars to int or Integer

        String stringNum = "3";

        i = Integer.parseInt (stringNum);           //i has 3
        ref = Integer.valueOf  (stringNum);        //ref to 3

        p ("3. from String to i=" + i + " or ref=" + ref);

/*4*/   //Convert int or Integer to String with digit chars

        stringNum = Integer.toString (4);     //static method
        stringNum = ref.toString();         //instance method

        p ("4. from int or Integer to String=" + stringNum);

/*5*/   double d = ref.doubleValue();//see javadoc for Number
        p ("5. from Integer to any numeric basic type=" + d);

/*6*/   Integer ref6 = new Integer (6);
        if ( ref.equals(ref6) ) //test same class & same data
            p ("6. two Integer objects with the same int");
    }

    public static void p (String s) {
        System.out.println (s);
    }
}
```

Result, AJ515.java

```
1. i=1, r=1
2. fits in Integer: 2
3. from String to i=3 or ref=3
4. from int or Integer to String=3
5. from Integer to any numeric basic type=3.0
```


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 (these four examples use Locale.US):

   a.
   ```
   NumberFormat a = NumberFormat.getInstance ();
   System.out.println(a.format(12345.12345));
   //12,345.123
   ```

   b.
   ```
   NumberFormat b = NumberFormat.getNumberInstance ();
   System.out.println(b.format(12345.12345));
   //12,345.123
   ```

   c.
   ```
   NumberFormat c = NumberFormat.getCurrencyInstance ();
   System.out.println(c.format(12345.12345));
   //$12,345.12
   ```

   d.
   ```
   NumberFormat d = NumberFormat.getPercentInstance ();
   System.out.println(d.format(25));
   //2,500%
   System.out.println(d.format(.3456));
   //35%    //rounded
   ```

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 the facing page. The Locale class is in the java.util package.

EDITING NUMBERS EXAMPLE

AJ517.java

```
import java.text.NumberFormat;
import java.util.Locale;

public class AJ517 {
    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, AJ517.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

AJ518.java

```
import java.text.NumberFormat;
import java.util.Locale;

public class AJ518 {

    public static final int COLUMN_WIDTH = 16;

    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 = COLUMN_WIDTH - n.length();

        StringBuilder sb = new StringBuilder ();
        for (int i=1; i<=spacesNeeded; i++) {
            sb.append(' ');           //append leading spaces
        }
        sb.append(n);                 //append number String

        System.out.println (":123456789-123456:ruler line");
        System.out.println (":" + sb + ":");
    }
}
```

Result, AJ518.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.

FOR-EACH LOOP, ADDED IN JAVA 5

AJ519.java

```
public class AJ519 {
    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;

        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) {
            System.out.print (abbreviation + " ");
        }
        System.out.println ();
    }
}
```

Result, AJ519.java

```
Maine Ohio Alaska NY NJ null null
```

==

1. for (dataTypeOfArrayElement nameForTempVar : arrayName)
   statement_loopBodyExecutedOnceForEachArrayElement;

2. The for-each 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 first word in parentheses is the data type of the elements in the array or collection.

4. The word after the colon is the name of the array or collection to be processed.

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.

6. An array, collection, or any class type that implements the interface Iterable can be processed.

EXERCISES

1. Copy CaseStudy4.java and RoomReservation4.java, and call the copies CaseStudy5.java and RoomReservation5.java.

   <u>CaseStudy5.java in com.themisinc.u05</u>

   a. Create an array of RoomReservation5 type, and populate the array with RoomReservation5 objects.

   b. Use a foreach loop to call the printOneReservation method of each object in the array.

   <u>RoomReservation5.java in com.themisinc.u05</u>

   c. Modify the setReservationNumber method to convert the int reservationNumber to a String and validate it according to the requirements below. Use a StringBuilder to create the error message, and if there are errors then print to the console via System.err.println and use the default reservation number 130789.

      1) The String length must be 6.

      2) The first three characters must be "130".

      3) The fourth, fifth, and sixth characters must not be the same. For example, the reservation number 130444 is invalid because of the 444.

   d. Run your program several times with different invalid reservation numbers to test your code.

   e. Create a method called formatMoney to format a dollar amount for printing with two decimal places. The method receives one double parameter to be formatted, and returns a 12-character String with the formatted dollar amount right-justified with leading spaces. In the method use a private instance StringBuilder called sbMoney.

   f. Create a method called intTo12String that receives one int parameter and returns a 12-character String with the int value right-justified with leading spaces.

   g. Modify the method printOneReservation to call the method formatMoney to format dollar amounts, and the method intTo12String to format int values, before printing them.

   h. Modify the method printOneReservation to use a private instance StringBuilder to create the String to be printed.

SOLUTIONS

CaseStudy5.java in com.themisinc.u05

```
package com.themisinc.u05;
public class CaseStudy5 {
    public static void main (String[] args) {

        RoomReservation5[] rrArray = new RoomReservation5[2];

        rrArray[0] = new RoomReservation5 (
            130323, 12, 5, 25.00);
        rrArray[1] = new RoomReservation5 (
            1334445, 14, 3);                //invalid res no

        for (RoomReservation5 elem : rrArray) {
            if (elem != null)  {
                elem.printOneReservation();
            }
        }
    }
}
```

RoomReservation5.java in com.themisinc.u05

```
package com.themisinc.u05;
import java.text.NumberFormat;

public class RoomReservation5 {

    public static final int    DEFAULT_RESERVATION_NUMBER
        = 130789;
    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;

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

    private double roomAmount;

    private StringBuilder sb      = new StringBuilder();
    private StringBuilder sbMoney = new StringBuilder();
    private StringBuilder sbInt   = new StringBuilder();

    private NumberFormat nfMoney  =
        NumberFormat.getCurrencyInstance();

    public RoomReservation5 () {
    }
```



```
    public RoomReservation5 (
      int reservationNumber, int seats,
      int numberOfDays, double dayRatePerSeat) {
        setReservationNumber (reservationNumber);
        setSeats (seats);
        setNumberOfDays (numberOfDays);
        setDayRatePerSeat (dayRatePerSeat);
    }
    public RoomReservation5 (
      int reservationNumber,
      int seats,
      int numberOfDays
    ) {
        this (reservationNumber, seats,
            numberOfDays, DEFAULT_DAY_RATE_PER_SEAT);
    }

    private void calculateAmount () {
        roomAmount = seats * numberOfDays * dayRatePerSeat;
    }

    private String formatMoney (double d) {
        sbMoney.delete (0, sbMoney.length());
        sbMoney.append (nfMoney.format(d));
        int spacesNeeded = 12 - sbMoney.length();
        for (int i=1; i<=spacesNeeded; i++) {
            sbMoney.insert(0, ' ');
        }
        return sbMoney.toString();
    }
    private String intTo12String (int param) {
        sbInt.delete (0, sbInt.length());
        sbInt.append (Integer.toString (param));
        int spacesNeeded = 12 - sbInt.length();
        for (int i=1; i<=spacesNeeded; i++) {
            sbInt.insert(0, ' ');
        }
        return sbInt.toString();
    }

    public void printOneReservation () {
        calculateAmount ();
        sb.delete (0, sb.length());
        sb.append ("\nReservation:       ");
        sb.append (   intTo12String (reservationNumber) );
        sb.append ("\nNumber of seats:   ");
        sb.append (   intTo12String (seats) );
        sb.append ("\nNumber of days:    ");
        sb.append (   intTo12String (numberOfDays) );
        sb.append ("\nDay rate per seat: ");
        sb.append (   formatMoney(dayRatePerSeat));
        sb.append ("\nRoom amount:       ");
        sb.append (   formatMoney(roomAmount) + "\n");
```



```
        System.out.println (sb.toString());
    }

    public int getReservationNumber () {
        return reservationNumber;
    }
    public void setReservationNumber(int reservationNumber) {
        sb.delete(0, sb.length());
        String s = Integer.toString (reservationNumber);
/*1*/   if (s.length() != 6) {
            sb.append ("invalid length=");
            sb.append (s.length());
            sb.append ("\n");
        }
/*2*/   if (! s.startsWith ("130") ) {
            sb.append ("does not start with 130\n");
        }
/*3*/   char c3 = s.charAt (3);
        if (c3 == s.charAt(4) && c3 == s.charAt(5) ) {
            sb.append ("chars 4, 5, and 6 are the same\n");
        }
        if (sb.length() == 0) {
            this.reservationNumber = reservationNumber;
        } else {
            sb.insert (0, "\n");
            sb.insert (0, DEFAULT_RESERVATION_NUMBER);
            sb.insert (0, " is invalid, will use ");
            sb.insert (0, reservationNumber);
            sb.insert (0, "\n");
            System.err.println (sb.toString() );
            this.reservationNumber =
                DEFAULT_RESERVATION_NUMBER;
        }
    }

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

```

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

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

<u>Result, CaseStudy5.java in com.themisinc.u05</u>

```
1334445 is invalid, will use 130789
invalid length=7
does not start with 130
chars 4, 5, and 6 are the same


Reservation:             130323
Number of seats:             12
Number of days:               5
Day rate per seat:       $25.00
Room amount:          $1,500.00


Reservation:             130789
Number of seats:             14
Number of days:               3
Day rate per seat:       $25.00
Room amount:          $1,050.00
```

OPTIONAL EXERCISE, REVISIONS OF AJ518.java

Explain how the following six programs work, and what the differences would be in a production environment where you might have many transactions with many columns of numbers to format.

<u>**Result, all versions**</u>

```
:123456789-123456789:ruler
:        $12,345.68:
:    $12,345.68:
```

<u>**AJ518_rev1.java**</u>

```
import java.text.NumberFormat;  //ref USA has class scope.
import java.util.Locale;        //obj created when class is
public class AJ518_rev1 {       //loaded, exists to pgm exit.
    private static NumberFormat USA =
        NumberFormat.getCurrencyInstance (Locale.US);
    private static StringBuilder sb = new StringBuilder ();
    public static void main (String[] args) {
        System.out.println (":123456789-123456789:ruler");
        for (int i=18 ; i>10 ; i=i-4) {
            System.out.println(":" +form(12345.6789,i)+ ":");
        }
    }
    public static String form (double d, int width) {
        sb.delete (0, sb.length() );
        sb.append (USA.format(d) );
        int spacesNeeded = width - sb.length();
        for (int i=1;  i<=spacesNeeded;  i++) {
            sb.insert(0, ' ');    //append leading spaces
        }
        return sb.toString();
    }
}
```


AJ518_rev2.java

```
import java.text.NumberFormat; //ref USA is local in form().
import java.util.Locale;       //obj is created and gc'ed
public class AJ518_rev2 {      //for each call to form().
    private static StringBuilder sb = new StringBuilder ();
    public static void main (String[] args) {
        System.out.println (":123456789-123456789:ruler");
        for (int i=18 ; i>10 ; i=i-4) {
            System.out.println(":" +form(12345.6789,i)+ ":");
        }
    }
    public static String form (double d, int width) {
        NumberFormat USA =
            NumberFormat.getCurrencyInstance (Locale.US);
        sb.delete (0, sb.length() );
        sb.append (USA.format(d) );
        int spacesNeeded = width - sb.length();
        for (int i=1;  i<=spacesNeeded;  i++) {
            sb.insert(0, ' ');    //append leading spaces
        }
        return sb.toString();
    }
}
```

AJ518_rev3.java

```
import java.text.NumberFormat;  //ref USA is local in main()
import java.util.Locale;        //and must be arg to form().
public class AJ518_rev3 {       //obj created once if needed.
    private static StringBuilder sb = new StringBuilder ();
    public static void main (String[] args) {
        NumberFormat USA =
            NumberFormat.getCurrencyInstance (Locale.US);
        System.out.println (":123456789-123456789:ruler");
        for (int i=18 ; i>10 ; i=i-4) {
            System.out.println(
                ":" + form(USA, 12345.6789, i) + ":");
        }
    }
    public static String form (
      NumberFormat nf, double d, int width) {
        sb.delete (0, sb.length() );
        sb.append (nf.format(d) );
        int spacesNeeded = width - sb.length();
        for (int i=1;  i<=spacesNeeded;  i++) {
            sb.insert(0, ' ');    //append leading spaces
        }
        return sb.toString();
    }
}
```

AJ518_rev4.java

```
import java.text.NumberFormat;  //ref USA has class scope.
import java.util.Locale;        //obj is created once the
public class AJ518_rev4 {       //first time it is needed
    private static StringBuilder sb = new StringBuilder ();
    private static NumberFormat USA;
    public static void main (String[] args) {
        USA = NumberFormat.getCurrencyInstance (Locale.US);
        System.out.println (":123456789-123456789:ruler");
        for (int i=18 ; i>10 ; i=i-4) {
            System.out.println(":" +form(12345.6789,i)+ ":");
        }
    }
    public static String form (double d, int width) {
        sb.delete (0, sb.length() );
        sb.append (USA.format(d) );
        int spacesNeeded = width - sb.length();
        for (int i=1;  i<=spacesNeeded;  i++) {
            sb.insert(0, ' ');    //append leading spaces
        }
        return sb.toString();
    }
}
```

AJ518_rev5.java

```
import java.text.NumberFormat;  //ref USA is instance var in
import java.util.Locale;        //aj object. nf object exists
public class AJ518_rev5 {       //until aj object is gc'ed.
    private NumberFormat USA =
        NumberFormat.getCurrencyInstance (Locale.US);
    private StringBuilder sb = new StringBuilder ();
    public static void main (String[] args) {
        AJ518_rev5 aj = new AJ518_rev5();
        System.out.println (":123456789-123456789:ruler");
        for (int i=18 ; i>10 ; i=i-4) {
           System.out.println(":"+aj.form(12345.6789,i)+":");
        }
    }
    public String form (double d, int width) {
        sb.delete (0, sb.length() );
        sb.append (USA.format(d) );
        int spacesNeeded = width - sb.length();
        for (int i=1;  i<=spacesNeeded;  i++) {
            sb.insert(0, ' ');    //append leading spaces
        }
        return sb.toString();
    }
}
```


AJ518_rev6.java

```
import java.text.NumberFormat;  //This is the main class.
import java.util.Locale;        //The PrinterClass is after
public class AJ518_rev6 {        //the end of this class
    public static void main(String[] args) {
        PrinterClass6 p = new PrinterClass6();
        p.printRuler();
        for (int i = 18; i > 10; i = i - 4) {
            p.form(12345.6789, i);
        }
    }
}
```

PrinterClass6.java

```
public class PrinterClass6 {
    private NumberFormat USA =
            NumberFormat.getCurrencyInstance (Locale.US);
    private StringBuilder sb = new StringBuilder ();
    public void printRuler () {
        System.out.println(":123456789-123456789:ruler");
    }
    public void form (double d, int width) {
        sb.delete (0, sb.length() );
        sb.append (USA.format(d) );
        int spacesNeeded = width - sb.length();
        for (int i=1;  i<=spacesNeeded;  i++) {
            sb.insert(0, ' ');    //append leading spaces
        }
        sb.insert(0, ':');
        sb.append(':');
        System.out.println(sb);
    }
}
```

UNIT 6: CLASSES AND OBJECTS PART 2: INHERITANCE, ABSTRACT CLASSES, RUNTIME POLYMORPHISM, INTERFACES, PACKAGES AND import, final

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

1. Briefly explain:

    how inheritance is implemented in Java
    requirements and uses for overriding and overloading
    uses of the keyword super
    instanceof operator
    final classes, final methods, and final variables
    abstract classes and methods
    runtime polymorphism
    interfaces
    package and import directives

2. Create programs that make use of inheritance, overriding methods, the keywords super and final, the instanceof operator, abstract classes and methods, runtime polymorphism, interfaces, and package and import compiler directives.

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 API is organized into a big hierarchy of classes descending from the top superclass, Object in java.lang. 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. There must be a constructor in each ancestor that matches 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 arguments. A constructor that accepts no arguments may be called a "null constructor."

10. If a class is coded with no constructor javac gives it a default constructor that accepts no arguments and calls super with no arguments.

extends AND super(), EXAMPLE

AJ603.java

```
class I {
    private int i;
    public I (int i) {
        this.i = i;
        System.out.println ("class I has 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;
        System.out.println ("class J has j=" + j);
    }
    public int getIJSum () {
        return getI() + j;  //variable name i is out of scope
    }
}
public class AJ603 {
    public static void main (String[] args) {
        I refI = new I (1);
        J refJ = new J (10, 20);
        System.out.println ("I=" + refI.getI() +
                                  ", J=" + refJ.getIJSum() );
    }
}
```

Result, AJ603.java

```
class I has i=1
class I has i=10
class J has j=20
I=1, J=30
```

1. To properly initialize superclass variables, private or not, call super in the first statement in the subclass constructor. The arguments passed with super must be suitable for one of the immediate superclass's constructors.

2. Subclasses should not repeat the code in their ancestor classes. Those classes should validate, calculate, or initialize their own variables.

3. If inherited variables are private, they are not accessible by name and should be accessed via their public get and set methods.

OVERRIDING VERSUS OVERLOADING, super.METHOD()

1. A subclass inherits ALL the variables and methods of its entire ancestry, including private variables and methods.

    a. The identifiers of private inherited members are not in scope in the subclass.
    b. The identifiers of protected and public inherited members are in scope in the subclass.
    c. The identifiers of inherited members with no access modifiers are in scope in the subclass if the subclass is in the same package with the superclass.

2. A subclass can reference all the accessible variables and methods that it inherits, unless the subclass "hides" ("shadows") a superclass variable, or "overrides" a superclass method.

3. However, the keyword super can be used in an instance method to refer to an accessible hidden variable or accessible overridden method in the immediate superclass (the superclass may have coded or inherited it).

    a. The subclass hides (or shadows) a superclass variable by defining its own variable with the same name. However, the subclass can access a hidden accessible superclass variable via super.varName. This breaks encapsulation.

    b. The subclass overrides a superclass method by defining its own method with the same name, parameter list and return type. However, the subclass can access an overridden accessible superclass method via super.methodName().

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. The compiler javac 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, super.METHOD(), EXAMPLE

AJ605.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 {      //Line 17 overrides line 6 via
12      private int j1;       //same name, params, return type
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                                          //if super. is omitted
22  class J2 extends I {      //Line 28 overloads line 6 via
23      private int j2;       //same name, different params
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      }                                   //super. is not needed
31  }                                       //but self-documenting
32                                          //& helps readability
33  public class AJ605 {
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, AJ605.java

```
objI. 1
objJ1. 30
objJ2. 100
objJ2(5). 305
```


A SUBCLASS CAN BE USED AS SUPERCLASS TYPE BECAUSE IT ISA

1. A reference of a superclass type can point to an object of any of its subclasses.

    a. When you need a reference or object of a superclass type you may use a reference or object of its subclass instead. For example, a method that requires a parameter of a superclass type will accept a reference to one of its subclasses.

2. A subclass object ISA superclass object in the sense that the subclass object contains the same instance variables and methods in the same relative location in its object space.

a	getA( )		

a	getA( )	b	getB( )

3. The class type of a reference variable determines which identifiers are in scope within the pointed-to object.

    a. Problem: If a superclass reference points to a subclass object, only identifiers of the superclass are in scope to be accessed within the subclass object. How can you access the identifiers coded in the subclass?

    b. Solution: A reference of a superclass type can be cast to the subclass type to allow access to identifiers coded in the subclass.

    c. Warning: At runtime, the JVM verifies that the casted reference in fact points to an object of the casted subclass type, and throws an exception if it does not.

CAST A SUPERCLASS REFERENCE TO A SUBCLASS TYPE

AJ607.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 AJ607 {
    public static void main (String[] args) {

        int res1=0, res2=0, res3=0, res4=0, res5=0;

        I parentRef1 = new I(1);         //ref of type I has
        I parentRef2 = new J(2, 3);      //identifier getI()

        res1 = parentRef1.getI();    //getI() is coded method
        res2 = parentRef2.getI();//getI() is inherited method

      //res3 = parentRef2.getJ(); //child method not in scope

        if (parentRef2 instanceof J) {

            J childRef = (J) parentRef2; //cast ref to J type

            res3 =   childRef.getI() + childRef.getJ();
            res4 = parentRef2.getI() + childRef.getJ();

            res5 = ((J)parentRef2).getJ();//other way to cast
        }

        System.out.println(res1 + ", " + res2 + ", " + res3
            + ", " + res4 + ", " + res5);
    }
}
```

Result, AJ607.java

```
1, 2, 5, 5, 3
```


THE instanceof OPERATOR

<u>AJ608.java</u>

```
1   class I {
2   }
3
4   class J extends I {
5   }
6
7   class K extends J {
8   }
9
10  public class AJ608 {
11      public static void main (String[] args) {
12
13          I myI = new I ();
14          J myJ = new J ();
15          K myK = new K ();
16
17          if (myK instanceof I)                          //Wrong way
18              System.out.println ("1. myK points to I object");
19          else if (myK instanceof J)
20              System.out.println ("2. myK points to J object");
21          else if (myK instanceof K)
22              System.out.println ("3. myK points to K object");
23
24          if (myK instanceof K)                          //Right way
25              System.out.println ("4. myK points to K object");
26          else if (myK instanceof J)
27              System.out.println ("5. myK points to J object");
28          else if (myK instanceof I)
29              System.out.println ("6. myK points to I object");
30      }
31  }
```

<u>Result, AJ608.java</u>

```
1. myK points to I object
4. myK points to K object
```

==

1. The instanceof operator is used to determine the class of the object that a reference points to. The reference variable is coded on the left; the class name is coded on the right. The operator returns boolean.

2. The test is "<u>can the object pass as the specified type?</u>" All subclass objects can pass as their ancestors' types because subclasses inherit the members of their ancestors. Thus, instanceof returns true if asked if a reference to a subclass object points to an ancestor object. To determine the actual class of an object, you must ask about the subclass types in bottom-up sequence. Lines 24-29.

final CLASSES, METHODS, AND VARIABLES

AJ609.java

```
1   class I {
2       public static final int USEFUL_NUM = 123;     //final var
3       private int i;
4       public I (int i) {
5           this.i=i;
6       }
7       public final int getTot () {                //final method
8           return i;
9       }
10  }
11  final class J extends I {                        //final class
12      private int j;
13      public J (int i, int j) {
14          super(i);
15          this.j=j;
16      }
17      //public int getTot () {  } //can't override final method
18  }
19  public class AJ609 {
20      public static void main (String[] args) {
21
22          I objI = new I (1);
23          J objJ = new J (10, 20);
24
25          System.out.println ("useful=" + I.USEFUL_NUM +
26              ", objI.getTot=" + objI.getTot() +
27              ", objJ.getTot=" + objJ.getTot() );
28      }
29  }
```

Result, AJ609.java

```
useful=123, objI.getTot=1, objJ.getTot=10
```

1. If the keyword final is applied to a class, the class can not have 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, either in its declaration or later in a procedural statement. Line 2 above can be replaced by:

```
public static final int USEFUL_NUM;
USEFUL_NUM = 123;
```


ABSTRACT CLASSES AND METHODS ENFORCE STANDARDIZATION

1. It is problematic when many subclasses use different method names to do equivalent tasks. An abstract superclass can be used to enforce standardization of methodnames and functionality in all subclasses.

2. The compiler requires all abstract methods defined in an abstract superclass to be implemented (overridden) by all concrete (non-abstract) subclasses.

   a. A subclass that does not implement all abstract methods in its abstract superclasses must be declared abstract.

   b. Concrete subclasses may either contain or inherit concrete implementations of the abstract methods in their abstract superclasses. (A grandchild can inherit concrete implementations from its parent.)

3. A class is declared abstract by coding the keyword abstract in its header.

   a. A class that contains one or more abstract methods must be declared abstract.
   b. An abstract class may (but is not required to) contain abstract methods.
   c. An abstract class may (but is not required to) contain concrete methods.

4. An abstract method declaration:

   a. Must have the keyword abstract in its header.
   b. Must not have a body (the method header must be followed by ; semicolon rather than { } curly braces).
   c. Must be overridden by all concrete subclasses.
   d. Cannot be private or static, because private and static methods cannot be overridden.

5. An abstract class cannot be instantiated, but references of an abstract class type can be created to point to objects of its concrete subclasses.

6. An abstract superclass is part of the inheritance hierarchy, and its constructor will be called as part of the stack of constructor calls when a subclass object is created. If you do not code a constructor, Java provides a default one.

7. Often an abstract superclass is not specific or complete enough to be useful by itself, such as a BankAccount class that has SavingsAccount and CheckingAccount as subclasses.

ABSTRACT CLASSES AND METHODS, EXAMPLE

<u>AJ611.java</u>

```
abstract class Pet {                      //abstract class
    private String name;
    public Pet (String n) {
        name=n;
    }
    public String getName() {             //concrete method
        return name;
    }
    public abstract String getFavorite(); //abstract method
}

class Cat extends Pet {                   //Cat inherits one
    private String favoritePerch;         //concrete method,
    public Cat (String n, String f) {     //and must override
        super(n);                         //one abstract
        favoritePerch = f;                //method with a
    }                                     //concrete method.
    public String getFavorite() {
        return favoritePerch;
    }
}

class Dog extends Pet {                   //Dog inherits one
    private String favoritePlayArea;      //concrete method,
    public Dog (String n, String f) {     //and must override
        super(n);                         //one abstract
        favoritePlayArea = f;             //method with a
    }                                     //concrete method.
    public String getFavorite() {
        return favoritePlayArea;
    }
}

public class AJ611 {
    public static void main (String[] args) {

        Cat c = new Cat ("Kato", "waterheater");
        Dog d = new Dog ("Beau", "beach");

        System.out.println (
        c.getName() + " likes the " + c.getFavorite() +"\n"+
        d.getName() + " likes the " + d.getFavorite() );

    }
}
```

<u>Result, AJ611.java</u>

```
Kato likes the waterheater
Beau likes the beach
```


RUNTIME POLYMORPHISM IS TRIGGERED BY A PARENT REFERENCE TO A CHILD OBJECT, AND A CALL TO AN OVERRIDDEN METHOD

1. Runtime polymorphism means the JVM resolves a call to an overridden method at run time, based on the actual type of the object during runtime:

2. The two required triggers for runtime polymorphism are:
   a. A superclass reference points to a subclass object.
   b. Your code calls an overridden method (the superclass has the method and the subclass has overridden it).

   When your code calls the overridden method, the JVM performs the instanceof tests to determine the class of the subclass object, and calls the method of the subclass object.

3. Runtime polymorphism helps to create a simple, consistent interface to program functionality.

   a. A superclass can define the methods common to its subclasses, and the subclasses can implement their own procedures 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. Methods that are private, 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).

OPTIONAL NOTES

5. Runtime polymorphism is also called "dynamic method dispatch" or "virtual method invocation".

6. Two features of Java that allow Java to implement runtime polymorphism are:

   a. Method overriding, so that several methods have the same name, which represents the general functionality of the method. For example, most 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.

RUNTIME POLYMORPHISM, EXAMPLE

AJ613.java

```java
abstract class Pet {                      //abstract class
    private String name;
    public Pet (String n) {
        name=n;
    }
    public String getName() {             //concrete method
        return name;
    }
    public abstract String getFavorite(); //abstract method
}

class Cat extends Pet {                   //Cat inherits one
    private String favoritePerch;         //concrete method,
    public Cat (String n, String f) {     //and must override
        super(n);                         //one abstract
        favoritePerch = f;                //method with a
    }                                     //concrete method.
    public String getFavorite() {
        return favoritePerch;
    }
}

class Dog extends Pet {                   //Dog inherits one
    private String favoritePlayArea;      //concrete method,
    public Dog (String n, String f) {     //and must override
        super(n);                         //one abstract
        favoritePlayArea = f;             //method with a
    }                                     //concrete method.
    public String getFavorite() {
        return favoritePlayArea;
    }
}

public class AJ613 {
    public static void main (String[] args) {
        Pet[] a = new Pet [2];    //parent refs of type Pet
        a[0] = new Cat ("Gert",   "windowsill");
        a[1] = new Dog ("Woofie", "Union Square Dogrun");

        for (int i=0; i<a.length; i++) {
           System.out.println (a[i].getName() + " likes the "
           + a[i].getFavorite() );
        }
    }
}
```

Result, AJ613.java

```
Gert likes the windowsill
Woofie likes the Union Square Dogrun
```


INTERFACES FOR STANDARDIZATION OF METHODS

1. An interface defines a group of methods that have one specific purpose, such as handling taxes, commissions, mouse clicks on a button, or multi-threading.

    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 the interface method's 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 () ;
}
```

3. Up through Java 1.7, and interfaces could contain only constants and abstract methods.
    a. Variables in an interface are implicitly public, static, and final, so coding these modifiers is discouraged.
    b. Methods in an interface are implicitly public and abstract, so coding these modifiers is discouraged.

4. 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.

5. 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.

6. Interfaces have their own inheritance hierarchy. One interface can extend another to inherit its constants and methods, and may "hide" or "override" an inherited constant or method.

7. A public interface can be accessed by any class. A non-public interface can be accessed by classes in the same package.

8. 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.

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 instance vars
    private int agent;
    private double taxRate;

    public Policy () {                    //no-parameter ctor
    }
    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 + ",agent=" +
            agent + ",taxRate=" + taxRate;
    }

    public int    getAgent   () { return 0; }  //method stubs
    public void   setAgent   (int agent) {}
    public double getTaxRate () { return 0.0; }
    public void   setTaxRate (double taxRate) {}
}
```

AJ615.java

```
public class AJ615 {
    public static void main (String[] args) {
        Policy p = new Policy ( "615" );
        System.out.println (p);
    }
}
```

Result, AJ615.java

```
Policy:policyNo=615,agent=0,taxRate=0.0
```


INTERFACE REFERENCES SUPPORT RUNTIME POLYMORPHISM

AJ616.java
```
1   interface Breed {                        //Implementing classes
2       String getBreed() ;                  //must have these three
3       String getFavorite();                //methods.
4       String getName();                    //A breed reference knows
5   }
6                                        //these 3 method names.
7   class Pet {
8       private String name;                 //Subclasses will inherit
9       public Pet (String n) {              //name and getName()
10           name=n;
11      }
12      public String getName() {
13          return name;
14      }
15  }
16
17  class Cat extends Pet implements Breed {  //Must code methods
18      private String favoritePerch;         //getFavorite() and
19      public Cat (String n, String f) {     //getBreed()
20          super(n);                         //Will inherit name
21          favoritePerch = f;                //and getName()
22      }
23      public String getBreed() {            //overrides line 2
24          return "Tabby Cat";
25      }
26      public String getFavorite() {         //overrides line 3
27          return favoritePerch;
28      }
29  }
30
31  class Gerbil extends Pet implements Breed {
32      private String favoriteToy;
33      public Gerbil (String n, String f) {
34          super(n);
35          favoriteToy = f;
36      }
37      public String getBreed() {            //overrides line 2
38          return "Mongolian Gerbil";
39      }
40      public String getFavorite() {         //overrides line 3
41          return favoriteToy;
42      }
43  }
44
```



```
public class AJ616 {

    public static void main (String[] args) {

        Breed[] a = {
            new Cat ("Fluffy", "water heater"),
            new Gerbil("Gerbert", "running wheel")
        };

        for (Breed b : a) {
            System.out.println (
                b.getName() + ", " +
                b.getBreed() + ", likes the " +
                b.getFavorite()
            );
        }
    }
}
```

Result, AJ616.java

```
Fluffy, Tabby Cat, likes the water heater
Gerbert, Mongolian Gerbil, likes the running wheel
```

==

1. An interface will sometimes be designed to contain all the methods in a given group of subclasses so that an interface reference can be used to support runtime polymorphism, as shown in the example above.

2. An interface that lists all the methods to be coded in one or more classes to be developed later is often used during the early stages of project development. In such cases people use the term "coding to the interface."

3. Interfaces are covered in greater depth later in this course. Also, Java 1.8 introduced new interface features.

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 (aka folder). 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.

5. 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.

6. 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
    AJ619.java
    animals package
        Cat.java
        Dog.java
```

<u>AJ619.java in current package</u>

```java
public class AJ619 {
    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);
    }
 }
```

<u>Cat.java in package animals</u>

```java
package animals;
public class Cat {
     private String name;
     private String favoritePerch;
     public Cat (String n, String f) {
         name = n;
         favoritePerch = f;
     }
     public String toString() {
        return "Cat:" + name + "," + favoritePerch;
    }
}
```

<u>Dog.java in package animals</u>

```java
package animals;
public class Dog {
    private String name;
    private String favoritePlayArea;
    public Dog (String n, String f) {
        name = n;
        favoritePlayArea = f;
    }
    public String toString() {
        return "Dog:" + name + "," + favoritePlayArea;
    }
}
```

<u>Result, AJ619.java</u>

```
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 or load any class definitions, which occurs only if your code makes use of a class.

4. The import statement must be located after the package statement if there is one, and before any class declaration.

5. The scope of the import is from its location to the end of its compilation unit (in other words, its file).

6. Two ways to code import, so that your program can use either ClassName or packagename.ClassName

   a. import packagename.ClassName;

   b. import packagename.*;
      //javac and java will search packagename for classes and
      //interfaces used in your code 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.

9. Class names must be unique in each package.

THE import STATEMENT, EXAMPLE

AJ621.java in current package

```
import animals.Cat;                                            //new
import animals.Dog;                                            //new

public class AJ621 {
    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 f) {
         name = n;
         favoritePerch = f;
     }
     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 f) {
        name = n;
        favoritePlayArea = f;
    }
    public String toString() {
        return "Dog:" + name + "," + favoritePlayArea;
    }
}
```

Result, AJ621.java

```
Cat:Liberty,desk   Dog:Cisco,hall
```


READING EXERCISE

1. Read program E61.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).

E61.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 E61 {
    public static void main (String[] args) {
        Pet[] a = {
            new Cat ("Gert", "windowsill"),
            new Dog ("Woofie", "Union Square Dogrun"),
            new Fish ("Finney", "dried flies")
        };
        for (Pet p : a) {
            System.out.println (
            p.getName() + " likes the " + p.getFavorite() );
        }
    }
}
```


EXERCISES

1. Copy CaseStudy5.java and RoomReservation5.java, and call the copies CaseStudy6.java and RoomReservation6.java. In the new files change the name RoomReservation5 to RoomReservation6.

   In RoomReservation6 make the FormatMoney method public or protected so it can be called by subclass RoomResWithFood6. All files should be in the package com.themisinc.u06.

   Create RoomResWithFood6.java, a subclass of RoomReservation6, and FoodVendor6.java, a helper class for RoomResWithFood6.

   <u>FoodVendor6.java in com.themisinc.u06</u>

   a. The purpose of FoodVendor6 is to contain the names of the food vendor company and contact person.

   b. Create two private instance Strings, and public get and set methods for each of them. There is no validation for the Strings.

   ```
   companyName
   contact
   ```

   c. Create a constructor that accepts two Strings and calls the set methods to initialize the two variables.

   <u>RoomResWithFood6.java in com.themisinc.u06</u>

   d. The purpose of RoomResWithFood6 is to contain the food service requirement for a room reservation, calculate the cost, and print the details of the requirement.

   e. RoomResWithFood6 is a subclass of RoomReservation6, and has a helper class called FoodVendor6.

   f. Constants.

   ```
   public static final double AM_COST_PER_PERSON=9.00;
   public static final double PM_COST_PER_PERSON=8.00;
   ```

   g. Instance variables.

   ```
   private boolean amService;      with set and is methods
   private boolean pmService;      with set and is methods
   private double foodAmount;
   private FoodVendor6 fv;
   private StringBuilder sBldr = new StringBuilder();
   ```

h. Constructors.

   1) A null constructor that receives no parameters and has no statements.

   2) A constructor that receives eight parameters:
      int reservationNumber
      int seats
      int numberOfDays
      double dayRatePerSeat
      boolean amService
      boolean pmService
      String vendorCompany
      String vendorContact

      This constructor passes the first four parameters to super. The next two are passed to their set methods. The last two are used to construct a FoodVendor6 object.

   3) A constructor that receives seven parameters:
      int reservationNumber
      int seats
      int numberOfDays
      boolean amService
      boolean pmService
      String vendorCompany
      String vendorContact

      This constructor calls the constructor that receives eight parameters and passes:
      the first three parameters
      RoomReservation6.DEFAULT_DAY_RATE_PER_SEAT
      the last four parameters

i. A private void method called calculateAmount that determines whether AM and/or PM food service has been ordered, determines the food cost per person per day, and then multiplies by seats times numberOfDays to calculate the foodAmount.

j. A public void method called printOneReservation that calls the method calculateAmount of RoomResWithFood6, and then calls super.printOneReservation, and then prints additional lines with the food amount and the food vendor information.

<u>CaseStudy6.java in com.themisinc.u06</u>

k. Populate your RoomReservation6 array with several RoomReservation6 and RoomResWithFood6 objects. Use a loop to traverse the array and call the printOneReservation method for each object. Does runtime polymorphism occur? For which objects? How can you know?

2. Create interfaces for RoomResWithFood6.java and FoodVendor.java, as described below. Then modify the two classes to implement their appropriate interface, and execute CaseStudy6 again to make sure you get the same results as you got when you ran exercise 1.

a. The interface ReservationsWithFood.java should require implementing classes to have these methods:

```
printOneReservation
isAmService
setAmService
isPmService
setPmService
```

b. The interface Vendors.java should require implementing classes to have these methods:

```
getCompanyName
setCompanyName
getContact
setContact
```

c. The class header for RoomResWithFood6.java shown in the solutions does not have an implements clause. For this exercise the class header should be:

```
public class RoomResWithFood6
    extends RoomReservation6
    implements ReservationsWithFood {
```

d. The class header for FoodVendor6.java shown in the solutions does not have an implements clause. For this exercise the class header should be:

```
public class FoodVendor6 implements Vendors {
```

SOLUTIONS

CaseStudy6.java in com.themisinc.u06

```
package com.themisinc.u06;
public class CaseStudy6 {
    public static void main (String[] args) {

        RoomReservation6[] rrArray = {

            new RoomReservation6 (
                130323, 12, 5, 25.00),
            new RoomReservation6 (
                130445, 14, 3),

            new RoomResWithFood6 (
                130505, 12, 5, 45.00, true, true,
                "AB Food Services", "Arlene Banner"),
            new RoomResWithFood6 (
                130614, 14, 3, true, false,
                "CD Foods, Inc", "Charles Denrick"),
        };

        for (RoomReservation6 elem : rrArray) {
            if (elem != null)  {
                elem.printOneReservation();
            }
        }
    }
}
```

RoomReservation6.java in com.themisinc.u06 is the same as RoomReservation5.java except (1) package, class, and constructor names have 6 instead of 5 and (2) the method formatMoney has to be public or protected so it can be called by the subclass.

RoomResWithFood6.java in com.themisinc.u06

```
package com.themisinc.u06;
public class RoomResWithFood6 extends RoomReservation6 {

    public static final double AM_COST_PER_PERSON=9.00;
    public static final double PM_COST_PER_PERSON=8.00;

    private boolean amService;
    private boolean pmService;
    private FoodVendor6 fv;

    private double foodAmount;
    private StringBuilder sBldr = new StringBuilder();

    public RoomResWithFood6 () {
    }

```



```
    public RoomResWithFood6(
      int reservationNumber, int seats,
      int numberOfDays, double dayRatePerSeat,
      boolean amService, boolean pmService,
      String vendorCompany, String vendorContact
    ) {
        super (reservationNumber, seats,
             numberOfDays, dayRatePerSeat);
        setAmService (amService);
        setPmService (pmService);
        fv = new FoodVendor6 (vendorCompany, vendorContact);
    }

    public RoomResWithFood6(
      int reservationNumber, int seats,
      int numberOfDays,
      boolean amService, boolean pmService,
      String vendorCompany, String vendorContact
    ) {
        this (reservationNumber, seats,
        numberOfDays,
        RoomReservation6.DEFAULT_DAY_RATE_PER_SEAT,
        amService, pmService,
        vendorCompany, vendorContact);
    }

    private void calculateAmount () {
        double perDay = 0.0;
        if (isAmService() ) {
            perDay = AM_COST_PER_PERSON;
        }
        if (isPmService() ) {
            perDay = perDay + PM_COST_PER_PERSON;
        }
        foodAmount = getSeats() * getNumberOfDays() * perDay;
    }

    public void printOneReservation () {
        calculateAmount();
        super.printOneReservation();
        sBldr.delete (0, sBldr.length() );
        sBldr.append ("**Food Charges:");
        sBldr.append ("\n  Food:           ");
        sBldr.append (formatMoney(foodAmount) );
        sBldr.append ("\n  Vendor:         ");
        sBldr.append (fv.getCompanyName() );
        sBldr.append ("\n  Vendor contact: ");
        sBldr.append (fv.getContact() );
        sBldr.append ("\n");
        System.out.println (sBldr.toString() );
    }

```

```
    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;
    }
}
```

FoodVendor6.java in com.themisinc.u06

```
package com.themisinc.u06;
public class FoodVendor6 {
    private String companyName;
    private String contact;

    public FoodVendor6 (String companyName, String contact) {
        setCompanyName (companyName);
        setContact (contact);
    }

    public String getCompanyName () {
        return companyName;
    }
    public void setCompanyName (String companyName) {
        this.companyName = companyName;
    }

    public String getContact () {
        return contact;
    }
    public void setContact (String contact) {
        this.contact = contact;
    }
}
```

ReservationsWithFood.java

```
package com.themisinc.u06;
public interface ReservationsWithFood {
    void printOneReservation () ;
    boolean isAmService () ;
    void setAmService (boolean amServiceRequested) ;
    boolean isPmService () ;
    void setPmService (boolean pmServiceRequested) ;
}
```


<u>Vendors.java</u>

```
package com.themisinc.u06;
public interface Vendors {
    String getCompanyName () ;
    void setCompanyName (String companyName) ;
    String getContact () ;
    void setContact (String contactName) ;
}
```

<u>Result, CaseStudy6.java in com.themisinc.u06</u>

```
Reservation:               130323
Number of seats:               12
Number of days:                 5
Day rate per seat:         $25.00
Room amount:            $1,500.00


Reservation:               130445
Number of seats:               14
Number of days:                 3
Day rate per seat:         $25.00
Room amount:            $1,050.00


Reservation:               130505
Number of seats:               12
Number of days:                 5
Day rate per seat:         $45.00
Room amount:            $2,700.00

**Food Charges:
  Food:                 $1,020.00
  Vendor:          AB Food Services
  Vendor contact:  Arlene Banner


Reservation:               130614
Number of seats:               14
Number of days:                 3
Day rate per seat:         $25.00
Room amount:            $1,050.00

**Food Charges:
  Food:                   $378.00
  Vendor:          CD Foods, Inc
  Vendor contact:  Charles Denrick
```

REVIEW SUMMARY: CLASS, METHOD, VARIABLE

WHAT'S IN A CLASS?

```
[package statement;]
[import statements;]
[public] [abstract] [final] class ClassName
    [extends SuperClassName]
    [implements InterfaceName1, InterfaceName2]
    {
        static variables and methods;
        instance data members;
        constructors;
        instance method members;
}
```

WHAT'S IN A METHOD DECLARATION?

```
[public, protected, private] [static] [abstract] [final]
returnType methodName ( [type paramName1, type paramName2] )
    [throws Exception1, Exception2] {
        [statements in the method body]
        [return statement(s) required unless method is void]
}
```

WHAT'S IN A VARIABLE DECLARATION?

```
[public, protected, private] [static] [final] type variableName
[= initialValue] ;
```

1. Local Variables:
    a. Method parameters and variables defined inside a method.
    b. Needed only by that one method.
    c. Contain garbage until initialized by your code.
2. Instance Variables:
    a. Declared outside methods, often at the top of the class.
    b. Needed by multiple methods.
    c. Usually private.
    d. Instantiated inside each object of the class.
3. Static Variables:
    a. Declared with keyword static outside any method, often at the top of the class.
    b. Those with public, protected, or default access are accessible to code in other classes in your program.
    c. Instantiated in the static area, qualified by the classname, and not associated with any object.

(blank)

UNIT 7: DATES AND CALENDARS

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.

3. Use the Calendar class to convert Date information to "calendar fields" such as day of the week or year.

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. Instance methods to 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 parse a String that represents a date and determine what date it is, use the java.text.DateFormat class.

6. To calculate calendar values such as month, day of the week, or julian day of the year, use the java.util.Calendar class.

Date, DATES BEFORE OR AFTER, EXAMPLE

<u>**AJ703.java**</u>

```
import java.util.Date;

public class AJ703 {
    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);
    }
}
```

<u>**Result, AJ703.java**</u>

```
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. A java.text.ParseException is thrown if the String does not contain a recognizable date.

   b. The parse method only recognizes Strings containing the same DateFormat produced by the parse method's DateFormat object. On the facing page, on line 38, the String is in the DateFormat.SHORT format. On line 40, the reference dfS points to the object of DateFormat.SHORT that was created on lines 12-13. This is why the parse method can recognize the String's date.

Result, AJ705.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

AJ705.java

```
import java.util.Date;
import java.util.Locale;
import java.text.DateFormat;
import java.text.ParseException;

public class AJ705 {
    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 ("9. 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.

<u>Result, AJ707.java</u>

```
java.util.GregorianCalendar[time=1501940669435,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=2017,MONTH=7,WEEK_OF_YEAR=
31,WEEK_OF_MONTH=1,DAY_OF_MONTH=5,DAY_OF_YEAR=217,DAY_OF_WEEK=7,D
AY_OF_WEEK_IN_MONTH=1,AM_PM=0,HOUR=9,HOUR_OF_DAY=9,MINUTE=44,SECO
ND=29,MILLISECOND=435,ZONE_OFFSET=-18000000,DST_OFFSET=3600000]

2. today=Sat Aug 05 09:44:29 EDT 2017
3. add 31 days=Tue Sep 05 09:44:29 EDT 2017
4. set date=Sat Aug 12 10:09:29 EDT 2000
5. Mon Feb 25 00:00:00 EST 2013,Fri Mar 01 00:00:00 EST 2013
6. Mon,Fri
7. one-week reservation
```

Calendar, SimpleDateFormat, DATE ARITHMETIC, EXAMPLE

AJ707.java

```
import java.util.Calendar;
import java.util.Date;
import java.util.Locale;
import java.text.DateFormat;
import java.text.SimpleDateFormat;

public class AJ707 {
    public static void main (String[] args) throws Exception{


/*1*/   Calendar c = Calendar.getInstance(); //default is now
        System.out.println (c + "\n");


/*2*/   System.out.println ("2. today=" + c.getTime());


/*3*/   c.add (Calendar.DAY_OF_MONTH, 31);
        System.out.println ("3. add 31 days=" + c.getTime());


/*4*/   c.set (2000, 7, 12, 10, 9);
        System.out.println ("4. set date=" + c.getTime());


/*5*/   DateFormat df =
            DateFormat.getDateInstance (DateFormat.SHORT);
        Date start = df.parse ("2/25/13");           //Monday
        Date end   = df.parse ("3/1/13");            //Friday
        System.out.println ("5. " + start + "," + end);


/*6*/   SimpleDateFormat dow =              //E is day of week
            new SimpleDateFormat("E",Locale.US);
        String dowStart = dow.format (start);
        String dowEnd   = dow.format (end);
        System.out.println ("6. " + dowStart + "," + dowEnd);


/*7*/   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 ("7. one-week reservation");
        }
    }
}
```


OPTIONAL EXERCISES

1. Create a program called E71.java in com.themisinc.u07 that performs Date validation.

    a. Create a Date reference variable called startDate that points to a Date object holding a date of your choice.

    b. Create an int variable called numberOfDays and set it to a number between 1 and 5.

    c. Validate that the startDate is not a Saturday or Sunday.

    d. Validate that the startDate allows a course of the length specified in numberOfDays to be completed within Monday through Friday of the same week.

    e. Print a message to the console to indicate whether the startDate is valid or invalid.

OPTIONAL SOLUTIONS

E71.java in com.themisinc.u07

```
package com.themisinc.u07;

import java.util.Date;
import java.text.DateFormat;
import java.text.ParseException;
import java.util.Locale;
import java.text.SimpleDateFormat;

public class E71 {
    public static void main (String[] args)
    throws ParseException {

        int numberOfDays = 2;

        DateFormat dfs =
            DateFormat.getDateInstance(DateFormat.SHORT);

        Date startDate = dfs.parse ("5/16/13");   //Thursday

        //get a date formatter for day of the week
        SimpleDateFormat dow =            //E is day of week
            new SimpleDateFormat("E", Locale.US);

        //get a String with the name of the day, such as Mon
        String dowStr = dow.format(startDate);

        if (   (dowStr.equals("Mon") && numberOfDays < 6)
            || (dowStr.equals("Tue") && numberOfDays < 5)
            || (dowStr.equals("Wed") && numberOfDays < 4)
            || (dowStr.equals("Thu") && numberOfDays < 3)
            || (dowStr.equals("Fri") && numberOfDays < 2)
            ) {
            System.out.println ("OK startDate=" + startDate
            + "\nwith numberOfDays=" + numberOfDays);
        } else {
            System.err.println ("Bad startDate=" + startDate
            + "\nwith numberOfDays=" + numberOfDays);
        }
    }
}
```

Result, E71.java in com.themisinc.u07

```
OK startDate=Thu May 16 00:00:00 EDT 2013
with numberOfDays=2
```


(blank)

UNIT 8: EXCEPTIONS

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

1. Briefly describe exception handling, and use the keywords throw, throws, try, catch, and finally to handle exceptions.

2. Briefly describe the organization of standard exceptions, and which ones are required to be handled.

3. 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 from 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 in its own class, which must be a descendent of java.lang.Exception.

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

<u>MyException</u>

```
public class MyException extends Exception {
    public MyException () {
    }
    public MyException (String s) {
        super(s);
    }
}
```

<u>AJ803.java</u>

```
public class AJ803 {
    public static void main (String[]a) throws MyException {

        System.out.println ("1. main");

        try {
            throwMethod ('a');
        } catch (MyException e) {
            System.out.println ("3. catch, e=" + e);
        }

        System.out.println ("4. main");
        throwMethod ('c');
        throwMethod ('b');
    }

    static void throwMethod (char ch) throws MyException {

        System.out.println ("2. method called with " + ch);

        if (ch == 'a')
            throw new MyException ("a helpful message");
        if (ch == 'b')
            throw new MyException ();
    }
}
```

<u>Result, AJ803.java</u>

```
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 AJ803.throwMethod(AJ803.java:24)
        at AJ803.main(AJ803.java:14)
```


OPTIONAL: WAYS TO HANDLE EXCEPTIONS

1. An exception can be completely handled in the catch clause, that is, the code in your catch clause 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 <u>replaced</u> 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 <u>wrapped</u> 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);
    }
}
```

AJ805.java

```
public class AJ805 {

    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, AJ805.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 {
}
```

AJ806.java

```
public class AJ806 {
    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, AJ806.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. InstantiationException
         - c. NoSuchFieldException
         - d. NoSuchMethodException
         - e. <u>RuntimeException</u>
           - 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 <u>checked exception</u>.

   b. An exception that does not have to be acknowledged is called an <u>unchecked exception</u>.

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
```
1   public class MyException extends Exception {
2   }
```

AJ808.java
```
1   public class AJ808 {
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          try {
11              sub2();
12          } catch (MyException m) {
13              throw m;
14          }
15      }
16      public static void sub2() throws MyException {
17          sub3();
18      }
19      public static void sub3() throws MyException {
20          throw new MyException ();
21      }
22  }
```

Result, AJ808.java
```
MyException
        at AJ808.sub3(AJ808.java:20)
        at AJ808.sub2(AJ808.java:17)
        at AJ808.sub1(AJ808.java:11)
        at AJ808.main(AJ808.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.

EXERCISES

1. Create a program called E81.java in com.themisinc.u08.

    a. Create an Exception class called MyException.

    b. In the main class E81 in the main method, call a void method called method1 and pass an int which is either zero or non-zero.

    c. If method1 receives the int zero, method1 should return without throwing MyException. If method1 receives any other number, method1 should throw a MyException.

    d. In the main method, catch the MyException and print a message to say that it was caught. Use a finally clause so that regardless of whether or not main catches a MyException, you will print a message after the call to method1 to say that method1 was called.

2. READING EXERCISE Copy CaseStudy5.java and RoomReservation5.java, call the copies CaseStudy8.java and RoomReservation8.java, and put them in com.themisinc.u08.

    a. Create an Exception class called BadDataException.java in com.themisinc.u08 with two constructors, one null and one that accepts a String and passes it to super().

    b. In RoomReservation8, modify the set methods that validate seats, numberOfDays, and dayRatePerSeat so that instead of printing a message to the console they throw a BadDataException with a helpful message.

        1) The non-null constructors need throws clauses in their headers. The primary constructor should catch the Exception messages from the set methods and then throw them back to main in one Exception.

        2) Note: When a constructor throws an Exception, no object is created.

    c. In CaseStudy8, enclose your calls to RoomReservation8 constructors in try-catch-finally structures. In the catch clause, print the message from the Exception object. In the finally clause print "+ " followed by the input reservation number with no other text. Modify the data passed to the RoomReservation8 constructors so that each variable is specified with errors at least one time.

SOLUTIONS

<u>MyException.java in com.themisinc.u08</u>

```
package com.themisinc.u08;

public class MyException extends Exception {
}
```

<u>E81.java in com.themisinc.u08</u>

```
package com.themisinc.u08;

public class E81 {
    public static void main (String[] args) {

        try {
            method1 ( 0 );
        } catch (MyException m) {
            System.out.println ("caught MyException");
        } finally {
            System.out.println ("method1 was called");
        }
    }

    public static void method1(int n) throws MyException {
        if (n == 0) {
            return;
        }
        throw new MyException ();
    }
}
```

<u>Result, E81.java as shown above, passing 0 to method1</u>

```
method1 was called
```

<u>Result, E81.java passing 2 to method1</u>

```
caught MyException
method1 was called
```

<u>Note:</u>
To avoid warnings, put <u>@SuppressWarnings ("serial")</u> on the line above the class header in MyException.

BadDataException.java in com.themisinc.u08

```
package com.themisinc.u08;
public class BadDataException extends Exception {
    public BadDataException () {
    }
    public BadDataException (String s) {
        super(s);
    }
}
```

CaseStudy8.java in com.themisinc.u08

```
package com.themisinc.u08;
public class CaseStudy8 {
    public static void main (String[] args) {

        int rrn = 0;
        RoomReservation8[] rrArray = new RoomReservation8[2];

        try {
            rrn = 1303239;                  //invalid rrn
            rrArray[0] = new RoomReservation8 (
                rrn, 11, 0, 5.00);          //all invalid
        } catch (BadDataException e) {
            e.printStackTrace();
        } finally {
            System.out.println ("+ " + rrn + "\n");
        }

        try {
            rrn = 130445;
            rrArray[1] = new RoomReservation8 (
                rrn, 14, 8);                //invalid days
        } catch (BadDataException e) {
            e.printStackTrace();
        } finally {
            System.out.println ("+ " + rrn + "\n");
        }

        int i=1;
        for (RoomReservation8 elem : rrArray) {
            if (elem != null) {
                elem.printOneReservation();
            } else {
                System.out.println ("res " + i + " is null");
            }
            i++;
        }
    }
}
```


RoomReservation8.java in com.themisinc.u08

```
package com.themisinc.u08;
import java.text.NumberFormat;
public class RoomReservation8 {

    public static final int    DEFAULT_RESERVATION_NUMBER
        = 130789;
    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;

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

    private double roomAmount;

    private StringBuilder sb      = new StringBuilder();
    private StringBuilder sbMoney = new StringBuilder();
    private StringBuilder sbInt   = new StringBuilder();

    private NumberFormat nfMoney  =
         NumberFormat.getCurrencyInstance();

    public RoomReservation8 () {
    }
    public RoomReservation8 (
      int reservationNumber,
      int seats,
      int numberOfDays,
      double dayRatePerSeat) throws BadDataException {

        setReservationNumber (reservationNumber);

        StringBuilder sbCtor = new StringBuilder ();

        try {
            setSeats (seats);
        } catch (BadDataException e) {
            sbCtor.append (e.toString());
        }

        try {
            setNumberOfDays (numberOfDays);
        } catch (BadDataException e) {
            sbCtor.append (e.toString());
        }

        try {
            setDayRatePerSeat (dayRatePerSeat);
        } catch (BadDataException e) {
            sbCtor.append (e.toString());
        }
```

```

        if (sbCtor.length() > 0) {
             sbCtor.insert (0, ":\n");
             sbCtor.insert (0, reservationNumber);
             sbCtor.insert (0, "Reservation ");
             throw new BadDataException (sbCtor.toString());
        }
    }
    public RoomReservation8 (
      int reservationNumber,
      int seats,
      int numberOfDays)
        throws BadDataException
    {
        this (reservationNumber, seats,
            numberOfDays, DEFAULT_DAY_RATE_PER_SEAT);
    }

    private void calculateAmount () {
        roomAmount = seats * numberOfDays * dayRatePerSeat;
    }

    private String formatMoney (double d) {
        sbMoney.delete (0, sbMoney.length());
        sbMoney.append (nfMoney.format(d));
        int spacesNeeded = 12 - sbMoney.length();
        for (int i=1; i<=spacesNeeded; i++) {
            sbMoney.insert(0, ' ');
        }
        return sbMoney.toString();
    }
    private String intTo12String (int param) {
        sbInt.delete (0, sbInt.length());
        sbInt.append (Integer.toString (param));
        int spacesNeeded = 12 - sbInt.length();
        for (int i=1; i<=spacesNeeded; i++) {
            sbInt.insert(0, ' ');
        }
        return sbInt.toString();
    }

    public void printOneReservation () {
        calculateAmount ();
        sb.delete (0, sb.length());
        sb.append ("\nReservation:       ");
        sb.append (   intTo12String (reservationNumber) );
        sb.append ("\nNumber of seats:   ");
        sb.append (   intTo12String (seats) );
        sb.append ("\nNumber of days:    ");
        sb.append (   intTo12String (numberOfDays) );
        sb.append ("\nDay rate per seat: ");
        sb.append (   formatMoney(dayRatePerSeat));
        sb.append ("\nRoom amount:       ");
        sb.append (   formatMoney(roomAmount) + "\n");
        System.out.println (sb.toString());
    }

```



```
    public int getReservationNumber () {
        return reservationNumber;
    }
    public void setReservationNumber(int reservationNumber) {
        sb.delete (0, sb.length());
        String s = Integer.toString (reservationNumber);
/*1*/   if (s.length() != 6) {
            sb.append ("invalid length=");
            sb.append (s.length());
            sb.append ("\n");
        }
/*2*/   if (! s.startsWith ("130") ) {
            sb.append ("does not start with 130\n");
        }
/*3*/   char c3 = s.charAt (3);
        if (c3 == s.charAt(4) && c3 == s.charAt(5) ) {
            sb.append ("chars 4, 5, and 6 are the same\n");
        }
        if (sb.length() == 0) {
            this.reservationNumber = reservationNumber;
        } else {
            sb.insert (0, "\n");
            sb.insert (0, DEFAULT_RESERVATION_NUMBER);
            sb.insert (0, " is invalid, will use ");
            sb.insert (0, reservationNumber);
            sb.insert (0, "\n");
            System.err.println (sb.toString() );
            this.reservationNumber =
                DEFAULT_RESERVATION_NUMBER;
        }
    }

    public int getSeats () {
        return seats;
    }
    public void setSeats (int seats)
      throws BadDataException {
        switch (seats) {
            case 10: break;
            case 12: break;
            case 14: break;
            default: throw new BadDataException (
                     "  bad seats " + seats + "\n");
        }
        this.seats = seats;
    }

    public int getNumberOfDays () {
        return numberOfDays;
    }
    public void setNumberOfDays (int numberOfDays)
    throws BadDataException {
        if (numberOfDays < 1 || numberOfDays > 5) {
            throw new BadDataException (
            "  bad numberOfDays " +numberOfDays+ "\n");
        }
        this.numberOfDays = numberOfDays;
    }
```

```java

    public double getDayRatePerSeat () {
        return dayRatePerSeat;
    }
    public void setDayRatePerSeat (double dayRatePerSeat)
    throws BadDataException {
        if (dayRatePerSeat<25.00 || dayRatePerSeat>65.00) {
            throw new BadDataException (
            "  bad dayRatePerSeat " + dayRatePerSeat + "\n");
        }
        this.dayRatePerSeat = dayRatePerSeat;
    }
}
```

Notes:

1. System.out and System.err may arrive at the monitor screen in a different order. In Eclipse System.out is in black, System.err is in red.
2. If reservationNumber is invalid but other input values are valid, a RoomReservation object is created. If any other input values are invalid, an object is not created.
3. Whether or not an object is created, reservationNumber must be printed so the record can be located in the input data stream (whether from a file, database, or other source).
4. To avoid warnings, put @SuppressWarnings ("serial") on the line above the class header in BadDataException and MyException.

Result, CaseStudy8.java in com.themisinc.u08

```
1303239 is invalid, will use 130789
invalid length=7

com.themisinc.u08.BadDataException: Reservation 1303239:
com.themisinc.u08.BadDataException:   bad seats 11
com.themisinc.u08.BadDataException:   bad numberOfDays 0
com.themisinc.u08.BadDataException:   bad dayRatePerSeat 5.0

     at com.themisinc.u08.RoomReservation8.<init>(RoomReservatio
     n8.java:61)
     at com.themisinc.u08.CaseStudy8.main(CaseStudy8.java:11)
+ 1303239                                            ---System.out

com.themisinc.u08.BadDataException: Reservation 130445:
com.themisinc.u08.BadDataException:   bad numberOfDays 8

     at com.themisinc.u08.RoomReservation8.<init>(RoomReservatio
     n8.java:61)
     at com.themisinc.u08.RoomReservation8.<init>(RoomReservatio
     n8.java:71)
     at CaseStudy8.main(CaseStudy8.java:21)
+ 130445                                            ---System.out

res 1 is null                                       ---System.out
res 2 is null                                       ---System.out
```


(blank)

UNIT 9: 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. State which classes in java.io are node streams and wrapper streams. Wrap a node stream in a buffering wrapper.

5. Create a program that reads and writes ordinary disk files using byte streams.

A File OBJECT STORES A FILENAME

AJ902.java

```
import java.io.File;
public class AJ902 {
    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, AJ902.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 \\.

OPTIONAL: 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 console display, internal buffer, etc. 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	reference type
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. A node stream is an InputStream or OutputStream that connects to a source or destination of data, such as FileInputStream.

2. A wrapper stream is an InputStream that accepts another InputStream as a constructor argument, or an OutputStream that accepts another OutputStream as a constructor argument, such as BufferedInputStream or BufferedOutputStream.

   a. If you instantiate a FileInputStream, and pass its reference to the constructor of BufferedInputStream, your BufferedInputStream object can perform buffered input with the stream from the FileInputStream object. This is called <u>wrapping</u>, <u>chaining</u>,or <u>decorating</u> the FileInputStream in the BufferedInputStream.

3. PrintStream constructors were deprecated in Java 1.1 in favor of PrintWriters (optionally 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.

00	00	00	byte

FF	FF	FF	FF

   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

AJ906.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 AJ906 {
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, AJ906.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.

FILE STREAMS, COPY ONE RECORD AT A TIME

<u>AJ907.java</u>

```
import java.io.InputStream;
import java.io.FileInputStream;
import java.io.OutputStream;
import java.io.FileOutputStream;

public class AJ907 {

    private static final int RECORD_LENGTH = 10;

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

        int numBytesInRead=0;
        int byteCount=0;
        int recordCount=0;
        byte[] buf = new byte [RECORD_LENGTH];

        InputStream in = new FileInputStream ("in");
        OutputStream out = new FileOutputStream ("out");

        while (  (numBytesInRead = in.read(buf)) != -1) {

            //validate number of bytes read

            out.write (buf, 0, numBytesInRead);
            byteCount = byteCount + numBytesInRead;
            recordCount++;
        }

        in.close();
        out.close();
        System.out.println ("records=" + recordCount +
                            ", bytes=" + byteCount);
    }
}
```

<u>in</u>

```
aaaaa11111bbbbb22222ccccc33333ddddd44444     ---no newline at end
```

<u>Result, AJ907.java</u>

```
records=4, bytes=40
```

<u>out</u>

```
aaaaa11111bbbbb22222ccccc33333ddddd44444     ---no newline at end
```

==

1. Record-oriented files are typically 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 appears on the same line at the end of the file data.

BufferedInputStream, BufferedOutputStream

AJ908.java

```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 AJ908 {

    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 <u>bridge classes</u> because they wrap an InputStream or OutputStream and convert correctly between bytes and characters.

    a. PrintWriter is a wrapper that can wrap either an OutputStream or another Writer.

    b. The subclasses of Reader and Writer handle Unicode characters properly. The subclasses of InputStream and OutputStream that handle Unicode characters do so by using only the least significant 8 bits, which does not always represent the Unicode character correctly.

OPTIONAL: 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.

OPTIONAL: FileReader, FileWriter

<u>AJ911.java</u>

```
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 AJ911 {
    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();
    }
}
```

<u>data.txt (before)</u>

```
This is a data file to be copied.
```

<u>mycopy (before)</u>

```
Pre-existing data should be backed up.
```

<u>Result, AJ911.java</u>

```
Number of chars copied=34                     ---UNIX file length
```

<u>mycopy (after)</u>

```
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.

OPTIONAL: BufferedReader, BufferedWriter

AJ912.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 AJ912 {
    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.

OPTIONAL: BufferedReader, BufferedWriter, ANOTHER EXAMPLE

AJ913.java

```
1   import java.io.*;
2   public class AJ913 {
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") ) );
```

OPTIONAL: WRAP System.in

<u>AJ914.java</u>

```
1   import java.io.*;
2   public class AJ914 {
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  }
```

<u>Result, AJ914.java</u>

```
Enter a line: when the moon comes over the mountain
line=when the moon comes over the mountain, len=37
```

<u>Commandline execution</u>

```
$ java AJ914 < 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));
```

OPTIONAL: WRAP System.out

AJ915.java

```
1   import java.io.*;
2   public class AJ915 {
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, AJ915.java

```
Enter a line: When the moon comes over the mountain
line=When the moon comes over the mountain, len=37
```

Commandline execution

```
$ java AJ915 < 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.

OPTIONAL: BufferedInputStream WITH VARIABLE LENGTH RECORDS

AJ916.java

```
import java.io.*;
public class AJ916 {

    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, AJ916.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
```


EXERCISES

1. Create E91.java in com.themisinc.u09 and use OutputStream classes to create a file called reservations.txt to hold two or more records containing room reservation data. Assume that all data to be written to the file is already validated, and all fields contain valid data.

   a. Each record should have 16 ASCII characters (not Java chars) arranged in the following field positions:

   ```
   0-5    reservationNumber
   6-7    seats
   8-9    numberOfDays
   10-15  costPerSeatPerDay
   ```

   For example:

   ```
   0123456789-12345  ruler line
   1303231205025.00  record 1
   1304451403035.00  record 2
   1305051202045.00  record 3
   1306141401055.00  record 4
   ```

   b. Create your data as Strings, one String per record. You may use a String array. For each record, use a loop with the String method charAt to convert one char at a time to a byte, and assign the byte to one element of a byte array. Then write your byte array to the reservations.txt file.

2. READING EXERCISE

   CaseStudy9.java in com.themisinc.u09

   (You must execute E91 to obtain the file reservations.txt before you can execute CaseStudy9.java.) The main method creates a FileToArray9 object and calls its getArray method to get a reference to a RoomReservation5 array with data from reservations.txt. Then CaseStudy9.java prints the array data.

   FileToArray9.java in com.themisinc.u09

   The getArray method organizes the work of this class. First the input file length is validated to ensure that it is an exact multiple of the record size. The number of elements in rrArray is calculated by dividing file length by record length. Then reservations.txt is read one record at a time and each field is converted to int or double as needed to create a RoomReservation5 object. Then the object is created and its reference is assigned into the array.

   All reservation numbers are written to a log file.

SOLUTIONS

E91.java in com.themisinc.u09

```
package com.themisinc.u09;
import java.io.File;
import java.io.OutputStream;
import java.io.FileOutputStream;
import java.io.BufferedOutputStream;
public class E91 {

    public static final File OUTPUT_FILE_NAME =
        new File ("reservations.txt");
    public static final int RECORD_LENGTH = 16;

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

        //Output Record Layout
        //0-5    reservationNumber
        //6-7    seats
        //8-9    numberOfDays
        //10-15  costPerSeatPerDay

        String[] recArray = {
        //   0123456789-12345-----ruler line
            "1303231205025.00",
            "1304451403035.00",
            "1305051202045.00",
            "1306141401055.00",
        };

        OutputStream rrFile =
            new BufferedOutputStream (
                new FileOutputStream (OUTPUT_FILE_NAME));
        byte[] buf = new byte[RECORD_LENGTH];

        for (int rec=0; rec < recArray.length; rec++) {

            for (int ch=0; ch < RECORD_LENGTH; ch++) {
                buf[ch] = (byte) recArray[rec].charAt(ch);
            }
            rrFile.write (buf);
        }

        rrFile.close ();
    }
}
```

Result, E91.java -- There is no console output. File contents:

```
1303231205025.001304451403035.001305051202045.001306141401055.00
```

IN ECLIPSE reservations.txt WILL BE UNDER YOUR PROJECT. TO SEE IT IN PROJECT EXPLORER, HIGHLIGHT YOUR PROJECT NAME, RIGHT CLICK ON IT, THEN CLICK Refresh.

<u>CaseStudy9.java in com.themisinc.u09</u>

```
package com.themisinc.u09;

public class CaseStudy9 {

    public static void main (String[] args) {

        FileToArray9 fta = new FileToArray9 ();
        RoomReservation5[] rrArray = null;

        try {
            rrArray = fta.getArray ();
        } catch (Exception e) {
            e.printStackTrace ();
            System.exit (1);
        }

        if (rrArray == null) {
            System.err.println ("rrArray is null, exiting");
            System.exit (2);
        }

        for (int i = 0; i < rrArray.length; i++) {
            if (rrArray[i] != null) {
                rrArray[i].printOneReservation();
            } else {
                System.out.println ("res " + i + " is null");
            }
        }

    }
}
```

<u>RoomReservation5.java in com.themisinc.u09</u>

To copy RoomReservation5.java from com.themisinc.u05 to com.themisinc.u09:

1. Highlight the file in com.themisinc.u05
2. Right click on the highlighted file
3. In the popup, click Copy
4. Highlight com.themisinc.u09
5. Right click on the highlighted directory
6. In the popup, click Paste

FileToArray9.java in com.themisinc.u09

```
package com.themisinc.u09;

import java.io.File;
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 FileToArray9 {

    private File inFile = new File ("reservations.txt");
    private InputStream in;

    private File logFile = new File ("logOfRecordsRead.txt");
    private OutputStream log;

    private static final int RECORD_SIZE = 16;
    private byte[] buf = new byte[RECORD_SIZE];

    private RoomReservation5[] rrArray;
    private int nextSubscript = 0;

    public FileToArray9 () {
    }

    public RoomReservation5[] getArray () throws Exception {
        int arraySize = getArraySize();
        if (arraySize == 0) {
            return null;
        }

        rrArray = new RoomReservation5 [arraySize];
        readFilePopulateArray ();
        return rrArray;
    }

    private int getArraySize () {
        long inFileLength = inFile.length();

        if ((inFileLength % RECORD_SIZE) != 0) {
            System.err.println ("input file size error");
            System.exit(1);
        }
        return (int)(inFileLength / RECORD_SIZE);
    }

```



```
    private void readFilePopulateArray () throws Exception {
        if (rrArray == null) {
            System.err.println ("missing rrArray");
            System.exit(2);
        }

        in = new BufferedInputStream (
            new FileInputStream (inFile) );
        log = new BufferedOutputStream (
            new FileOutputStream (logFile) );
        int howManyBytesWereRead;
        int newline = '\n';

        while ((howManyBytesWereRead = in.read(buf)) != -1) {
            if (howManyBytesWereRead != RECORD_SIZE) {
                System.err.println (
                    "input read error, num bytes=" +
                    howManyBytesWereRead);
                System.exit(3);
            }
            log.write (buf, 0, 6);         //res num only
            log.write (newline);           //one per line
            createVariablesMakeObjAssignToArray ();
        }
        in.close();
        log.close();
    }

    private void createVariablesMakeObjAssignToArray ()
    throws Exception {
        int rrn;       //0-5    reservationNumber
        int seats;     //6-7    seats
        int days;      //8-9    numberOfDays
        double cost;   //10-15  costPerSeatPerDay

                                   //buf, startIndex, length
        rrn = Integer.parseInt   (new String (buf, 0, 6));
        seats = Integer.parseInt (new String (buf, 6, 2));
        days = Integer.parseInt  (new String (buf, 8, 2));
        cost = Double.parseDouble(new String (buf,10, 6));

        rrArray[nextSubscript] = new RoomReservation5 (
            rrn, seats, days, cost);
        nextSubscript++;
        return;
    }
}
```

Result, CaseStudy9.java

see next page

Result, CaseStudy9.java in com.themisinc.u09

```
Reservation:              130323
Number of seats:              12
Number of days:                5
Day rate per seat:        $25.00
Room amount:           $1,500.00


Reservation:              130445
Number of seats:              14
Number of days:                3
Day rate per seat:        $35.00
Room amount:           $1,470.00


Reservation:              130505
Number of seats:              12
Number of days:                2
Day rate per seat:        $45.00
Room amount:           $1,080.00


Reservation:              130614
Number of seats:              14
Number of days:                1
Day rate per seat:        $55.00
Room amount:             $770.00
```

IN ECLIPSE logOfRecordsRead.txt WILL BE UNDER YOUR PROJECT.
TO SEE IT IN THE PROJECT EXPLORER, HIGHLIGHT YOUR PROJECT NAME, RIGHT CLICK ON IT, THEN CLICK Refresh.

Result, CaseStudy9.java, in the file logOfRecordsRead.txt

```
130323
130445
130505
130614
```

Note:
To determine which input or output class would be best for a particular purpose, ask these questions:

1. Where does the data come from? file? database table? byte array? etc.
2. Is the data in ASCII bytes? EBCDIC bytes? Java chars? String?
3. Is the data already aggregated such as rows in a database table or records in a mainframe file, or does the data get created one transaction at a time such as from user input?

UNIT 10: INTRODUCTION TO JDBC

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

1. Use the JDBC classes and interfaces in the java.sql package to create and execute programs that will:

    Connect to a database
    Create a table
    Load data into the table
    Retrieve data from the table
    Delete a table

JDBC

1. JDBC stands for Java Data Base Connectivity. JDBC is an API consisting of JAVA interfaces and classes.

2. JDBC and associated drivers enable programs to connect to and manipulate tabular data, called the "data source," which can be a flat file, database management system, or relational database management system.

3. Except for database-specific non-standard syntaxes or options, JDBC enables Java programs to work in a vendor-independent and driver-independent way with databases such as MySQL, Access, Oracle, etc. JDBC enables programs to:

    a. connect to the data source.
    b. create and execute SQL statements such as updates or queries.
    c. retrieve and manipulate results received from queries.

4. JDBC provides the same functionality regardless of driver, so that switching of databases does not require code changes.

OPTIONAL

5. JDBC has several components:

    a. JDBC API - classes and interfaces in packages java.sql and javax.sql. The same API is in both the Java SE (Software Edition) and EE (Enterprise Edition). Basic classes and interfaces are in java.sql. The extension javax.sql has advanced capabilities such as Connection Pooling and Row Sets that are often needed for Java EE.

    b. The <u>DriverManager</u> class and <u>DataSource</u> interface provide the basic service for managing JDBC drivers.

    c. The <u>JDBC Test Suite</u> is a group of programs that test drivers for the necessary functionality.

    d. The <u>JDBC-ODBC Bridge</u> is software that converts JDBC method calls into ODBC function calls. Open Database Connectivity (ODBC) is a standard software interface for accessing databases that is independent of programming languages, database systems, and operating systems. Any application can use ODBC to query a database, regardless of the application's platform or the database it uses. The ODBC driver acts as a translation layer between the application and the database. The application only needs to know ODBC syntax, and the driver passes the query to the database in its native format, returning the results in a format the application can understand.

STEPS TO ACCESS A DATABASE (AKA DATA SOURCE)

1. Connect to the database
    a. Load the driver
    b. Connect to the data source

2. Database manipulation
    a. Create SQL statements
    b. Send SQL statements to the database
    c. Retrieve query results

3. Close connection to data source

DRIVERS

1. A database driver is software that can communicate with a database. The driver transmits your program's SQL to the database, and returns the database's results to your program.

2. Driver code is usually in a zip or jar file. Vendor documentation specifies the fully-qualified driver name.

3. Typically drivers are loaded via the Class.forName method

   a. Class.forName ("com.mysql.jdbc.Driver");
   b. Class.forName ("net.ucanaccess.jdbc.UcanaccessDriver");
   c. Class.forName ("com.ibm.db2.jdbc.app.DB2Driver");

4. Calling Class.forName(driverName) causes the class loader to load the driver, and the DriverManager class to "register" the driver (put the driver in a DriverManager's list).

5. The DriverManager class maintains a list of all registered drivers and gets a connection to one of them by offering the URL passed to getConnection to each driver until one of them recognizes the database server that the URL represents, and connects to it.

6. A Connection object represents a session with a specific database. SQL statements are executed and results are returned "within the context of a Connection."

OPTIONAL

7. Since Java 1.6 and JDBC4, DriverManagers load and register drivers found in the CLASSPATH. Applications that continue to explicitly load JDBC drivers via Class.forName() will continue to work without modification.

8. Database URLs have the format jdbc:subprotocol:subname.
   a. <u>jdbc</u> is the main protocol (equivalent to http)
   b. <u>subprotocol</u> designates the specific database management system, such as mysql
   c. <u>subname</u> provides additional information and can specify a data source. MySQL's format is //hostname:port/dbname
      1) jdbc:mysql://localhost:3306:customers ---on the localhost at MySQL's default port of 3306, connect to data source customers
      2) jdbc:odbc:ColorDB ---connect to ColorDB in Access
      3) jdbc:odbc://www.tahommel.com:5667:/cust/customers ---the host system is www.tahommel.com, 5667 is the port to use for the socket, and /cust/customers is the path to the database
      4) jdbc:db2:customers ---customers is the data source, a database alias that refers to the DB2 catalog entry on the DB2 client.

CONNECT TO Access DATABASE VIA DriverManager, EXAMPLE

AJ1005.java
```
1   import java.sql.Connection;
2   import java.sql.DriverManager;
3
4   public class AJ1005 {
5       public static void main (String[] args) {
6
7           String driver =
8             "net.ucanaccess.jdbc.UcanaccessDriver";
9          String dbURL =
10            "jdbc:ucanaccess://c:/myjava/database/Java2DB.mdb";
11
12          try {
13
14            Class.forName (driver);
15            Connection con=DriverManager.getConnection (dbURL);
16            System.out.println ("Connection=" + con);
17            con.close();
18
19          } catch ( ClassNotFoundException e ) {
20              e.printStackTrace();            //Class.forName()
21          } catch ( java.sql.SQLException e ) {
22              e.printStackTrace();            //getConnection()
23          }
24      }
25  }
```

Result, AJ1005.java
```
Connection=net.ucanaccess.jdbc.UcanaccessConnection@1de635a[c:\my
java\database\Java2DB.mdb]
```

===

1. The DriverManager's static method getConnection opens a connection to the specified database URL. Above, the Access database is Java2DB.mdb. Line 10 shows the full pathname.

OPTIONAL

2. Connecting to MySQL can require a username and password in addition to the dbURL.

3. Before Java 1.8, ODBC drivers were part of the JDK download:
```
String driver = "sun.jdbc.odbc.JdbcOdbcDriver";
String dbURL  = "jdbc:odbc:Java2DB";
```

4. Without UCanAccess, in Windows systems a DSN (Data Source Name) has to be set up before you can access the database.

createStatement METHOD OF Connection INTERFACE
executeUpdate METHOD OF Statement INTERFACE

1. A Connection object can be used to create statements of three different interface types to send SQL statements (aka queries) to the database:

   a. Statement, used to send a single query that will not have to be repeated, typically a query without parameters.

   b. PreparedStatement, for queries that are used repeatedly with different parameters (see aj10.14-aj10.15).

   c. CallableStatement, used to invoke a stored procedure, discussed in optional unit 11.

2. Statement stmt = con.createStatement(); //lines 14 and 20

   a. createStatement() is a Connection instance method and requires a Connection object connected to the database.

   b. The method returns a default Statement object that implements the Statement interface and can send SQL statements to the database.

3. stmt.executeUpdate(String SQL);   //lines 22-25 and 29

   a. executeUpdate(String SQL) is an instance method of the Statement interface.

   b. Executes the SQL statement, which may be an INSERT, UPDATE, DELETE, or an SQL statement that returns nothing (can not be a statement that returns a ResultSet).

   c. Returns either the int row count for SQL DML (Data Manipulation Language) statements that return rows, or 0 for SQL statements that return nothing.

   d. Throws an SQLException if a database access error occurs, or the method is called on a closed Statement, or the SQL statement produces a ResultSet object.

   e. On the facing page on lines 22-25, the stmt is used to call the executeUpdate method and send SQL to the database to create a table called Orders with three columns per row (which are a string, int, and double). On line 29 the stmt is used to drop (delete) the table.

4. Closing your Connection releases all resources for the Connection, and Statements and ResultSets related to it.

CREATE AND DROP A TABLE, EXAMPLE

AJ1007.java

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
import java.sql.SQLException;

public class AJ1007 {
  public static void main(String[] args) throws SQLException{

    String driver =
      "net.ucanaccess.jdbc.UcanaccessDriver";
    String dbURL =
      "jdbc:ucanaccess://c:/myjava/database/Java2DB.mdb";
    Connection con = null;
    Statement stmt = null;

    try {

      Class.forName (driver);
      con = DriverManager.getConnection (dbURL);
      stmt = con.createStatement();

      int ret = stmt.executeUpdate ("CREATE TABLE Orders (" +
          "color VARCHAR(20), "  +
          "gallons INT, "        +
          "price DOUBLE);"       );

      System.out.println ("1. CREATE TABLE, ret=" + ret);

      ret = stmt.executeUpdate ( "DROP TABLE Orders;" );

      System.out.println ("2. DROP TABLE, ret=" + ret);

    } catch ( ClassNotFoundException e ) {
        e.printStackTrace();
    } catch ( java.sql.SQLException e ) {
        e.printStackTrace();
    } finally {
        try {
            if (con != null)  con.close();
        } catch (SQLException e) {
            //handle it
        }
    }
  }
}
```

Result, AJ1007.java

```
1. CREATE TABLE, ret=0
2. DROP TABLE, ret=0
```


executeUpdate AND executeQuery METHODS OF Statement

1. SQL statements can be sent to a database, and the results retrieved, by the Statement methods executeQuery and executeUpdate. Both methods can throw SQLException.

   a. int executeUpdate (String SQL); ---see aj10.06

   b. ResultSet executeQuery (String SQL); Executes SQL by sending it to the database, and returns SQL's results in a ResultSet object containing zero or more rows.

2. Only one ResultSet object per Statement object can be open at the same time.

   a. Problem: If you request a new ResultSet, and your Statement already has a ResultSet open, your Statement will close the old ResultSet.

   b. Solution: If reading one ResultSet must be interleaved with reading a second, use a separate Statement object to obtain each ResultSet.

3. On the facing page:

   a. Lines 24-29 show SQL INSERT statements.

   b. Line 31 selects (retrieves) all rows from the table Orders into a ResultSet object called r.

   c. Lines 33-37 iterate while r has a next row, so columns 1-3 can be printed from each row via these ResultSet methods:

method	returns
r.next()	true if cursor can move to next row
r.getRow()	current row number as an int
r.getString(1)	String value of column 1 in current row
r.getInt(2)	int value of column 2 in current row
r.getDouble(3)	double value of column 3 in current row

4. The order of rows in a database table is indeterminate, and is not related to the order of rows in a ResultSet.

INSERT INDIVIDUAL ROWS, SELECT *, EXAMPLE

AJ1009.java

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
import java.sql.ResultSet;

public class AJ1009 {
  public static void main(String [] args) {

    String driver =
      "net.ucanaccess.jdbc.UcanaccessDriver";
    String dbURL =
      "jdbc:ucanaccess://c:/myjava/database/Java2DB.mdb";

    try {
      Class.forName (driver);
      Connection con=DriverManager.getConnection (dbURL);
      Statement stmt = con.createStatement();

    //stmt.executeUpdate ( "DROP TABLE Orders;" );
      String create="CREATE TABLE Orders " +
          "(color VARCHAR(20), gallons INT, price DOUBLE);" ;
      stmt.executeUpdate (create);

      stmt.executeUpdate( "INSERT INTO Orders VALUES " +
          "('green ', 1, 1.10);"  );
      stmt.executeUpdate( "INSERT INTO Orders VALUES " +
          "('red   ', 2, 2.20);"  );
      stmt.executeUpdate( "INSERT INTO Orders VALUES " +
          "('blue  ', 3, 3.30);"  );

      ResultSet r=stmt.executeQuery("SELECT * FROM Orders;");

      while (r.next() ) { //r.next() moves cursor to next row
          System.out.println (
             r.getRow()  + ".  " +  r.getString(1) + "\t" +
             r.getInt(2) + "\t"  +  r.getDouble(3)  );
      }

      if (con != null)  con.close();

    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Result, AJ1009.java

```
1.  green       1       1.1
2.  red         2       2.2
3.  blue        3       3.3
```


Connection METHOD CreateStatement, ResultSet AND CURSOR

1. Rows returned from a query are called the result set and are held in an object that implements the ResultSet interface.

2. A ResultSet has a cursor (a pointer to a specific row). Moving the cursor allows loops to iterate over the rows, as shown on the previous page, lines 33-37.

3. Statements can be created with parameters to specify the type and concurrency for any ResultSet the Statement returns.

    a. Type specifies scrolling and sensitivity:

        1) Scrolling is a ResultSet cursor's ability to move forward only, or forward and backward.
        2) Sensitivity controls whether the ResultSet can change due to concurrent changes in the underlying database.

    b. Types are:

        1) ResultSet.TYPE_FORWARD_ONLY, scroll forward only, not sensitive.
        2) ResultSet.TYPE_SCROLL_INSENSITIVE, scroll forward or backward, not sensitive.
        3) ResultSet.TYPE_SCROLL_SENSITIVE, scroll forward or backward, and the ResultSet can change due to concurrent changes in the underlying database.

    c. Concurrency controls whether a ResultSet is readonly or updatable. This unit covers ResultSet.CONCUR_READ_ONLY. Updating is in optional Unit 11.

4. Overloaded Connection method createStatement:

    a. createStatement() accepts no parameters and returns a default Statement object. ResultSets returned will have a cursor for scrolling TYPE_FORWARD_ONLY and concurrency CONCUR_READ_ONLY.

    b. createStatement (int type, int concurrency) enables you to create a Statement for which you can select the scrolling type and concurrency for ResultSets.

5. Methods to move a scrollable cursor: ---see also aj10.12

    a. boolean relative (int r); Moves cursor r rows. If r is negative move backward toward row 1.

    b. boolean absolute (int r); Moves cursor to row r. The first row is 1. If r is negative the absolute row is counted backward from the end of the ResultSet, so absolute(-1) puts the cursor on the last row.

createStatement (int type, int concurrency), EXAMPLE

<u>AJ1011.java</u>

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
import java.sql.ResultSet;

public class AJ1011 {
  public static void main(String [] args) {

    String driver =
        "net.ucanaccess.jdbc.UcanaccessDriver";
    String dbURL =
        "jdbc:ucanaccess://c:/myjava/database/Java2DB.mdb";

    try {
      Class.forName (driver);
      Connection con=DriverManager.getConnection (dbURL);
      Statement stmt = con.createStatement (
          ResultSet.TYPE_SCROLL_INSENSITIVE,
          ResultSet.CONCUR_READ_ONLY );

      ResultSet r=stmt.executeQuery("SELECT * FROM Orders");

      r.absolute (3);
      System.out.println (r.getRow() + "=" + r.getString(1));

      r.relative (-1);
      System.out.println (r.getRow() + "=" + r.getString(1));

      r.absolute (1);
      System.out.println (r.getRow() + "=" + r.getString(1));

      if (con != null) con.close();

    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

<u>Result, AJ1011.java</u>

```
3=blue
2=red
1=green
```


ResultSet METHODS TO MOVE THE CURSOR

1. When a ResultSet is created, the cursor is before row 1. The first call to next() moves the cursor to row 1. This enables loops to start with the next() method, and use its boolean return value to control iteration. Successive calls to next() move the cursor one row at a time through all rows.

2. The method next() can be used with any ResultSet regardless if scrollable or not. It returns true if the cursor is on a row, or false if it is after the last row.

3. The cursor can be moved by many methods if the ResultSet is scrollable. These methods throw exceptions.

   a. boolean next(); Moves cursor forward one row.
   b. boolean previous(); Similar to next() but goes backward.
   c. boolean first(); Moves cursor to row 1.
   d. boolean last(); Moves cursor to last row.
   e. boolean beforeFirst(); Moves cursor to before row 1.
   f. boolean afterLast(); Moves cursor to after the last row.
   g. boolean relative (int r); Moves cursor r rows. If r is negative the cursor moves backward toward row 1. Thus relative(1) and next() do the same. relative(-1) and previous() do the same.
   h. boolean absolute (int r); Moves cursor to row r. The first row is 1, so absolute(1) and first() do the same. If r is negative the absolute row is counted backward from the end of the ResultSet, so absolute(-1) is the same as last().

4. ResultSet methods to determine the cursor position include:

   a. int getRow() returns the int number of the current row.
   b. boolean isFirst() returns true if cursor is on first row.
   c. boolean isLast() returns true if cursor is on last row.
   d. boolean isBeforeFirst() returns true if cursor is before the first row.
   e. boolean isAfterLast() returns true if cursor is after the last row.

MOVE THE ResultSet CURSOR, EXAMPLE

AJ1013.java

```
import java.sql.*;
public class AJ1013 {
  public static void main(String [] args) {

    String driver =
      "net.ucanaccess.jdbc.UcanaccessDriver";
    String dbURL =
      "jdbc:ucanaccess://c:/myjava/database/Java2DB.mdb";

    try {
      Class.forName (driver);
      Connection con=DriverManager.getConnection (dbURL);
      Statement stmt = con.createStatement (
          ResultSet.TYPE_SCROLL_INSENSITIVE,
          ResultSet.CONCUR_READ_ONLY );

      ResultSet r=stmt.executeQuery("SELECT * FROM Orders;");

      r.last();     //maybe an error! loop will skip last row
      while (r.previous() ) {
          p ("1. " + r.getRow() + "=" + r.getString(1));
      }

      r.afterLast();      //loop will start with the last row
      while (r.previous() ) {
          p ("2. " + r.getRow() + "=" + r.getString(1));
      }

      if ( r.isFirst() )       p ("3. first");
      if ( r.isBeforeFirst() ) p ("4. before first");

      if (con != null)  con.close();

    } catch (Exception e) {
        e.printStackTrace();
    }
  }
  public static void p (String s) {
    System.out.println (s);
  }
}
```

Result, AJ1013.java

```
1. 2=red
1. 1=green
2. 3=blue
2. 2=red
2. 1=green
4. before first
```


PreparedStatement

1. A PreparedStatement object is returned by the Connection method prepareStatement. By default, ResultSets created by a PreparedStatement are TYPE_FORWARD_ONLY and CONCUR_READ_ONLY but different type and concurrency can be specified when you call the method prepareStatement.

2. A PreparedStatement must be provided with an SQL String when it is created, rather than when it is executed.

    a. The SQL will be sent to the database management system to be compiled immediately if the driver and server support precompilation.

    b. Precompilation can result in faster execution.

3. PreparedStatements are typically used with SQL with <u>parameters</u> so that the same SQL can be re-executed with different values.

    a. PreparedStatements do not require use of parameters.

4. The SQL string can contain one or more ? question marks as <u>parameter placeholders</u> which should be replaced by <u>substitution values</u> before the prepared statement is executed, as shown on the facing page on lines 14-21.

    a. A value is <u>bound</u> to a parameter by a set method for the SQL type of the column that the parameter is for.

    b. Parameter values remain in force for repeated use of a PreparedStatement. Setting a parameter value clears its previous value.

    c. To immediately clear parameter values, call the instance method clearParameters. This is useful to immediately release resources used by the current parameter values.

5. An SQLException is thrown if the parameters have not been bound to values before execution.

6. SQL statements with parameters are called "dynamic SQL" because they can be executed repeatedly with different values.

7. Parameters are often used with SQL insert statements, but can be used with others: "DELETE FROM Orders WHERE color = ?"

PreparedStatement, EXAMPLE

<u>**AJ1015.java**</u>

```
import java.sql.*;
public class AJ1015 {
  public static void main(String [] args) {

    String driver =
      "net.ucanaccess.jdbc.UcanaccessDriver";
    String dbURL =
      "jdbc:ucanaccess://c:/myjava/database/Java2DB.mdb";

    try {
      Class.forName (driver);
      Connection con=DriverManager.getConnection (dbURL);

      String sql = "INSERT INTO Orders VALUES (?, ?, ?)";
      PreparedStatement pStmt = con.prepareStatement (sql);

      pStmt.setString (1, "yellow");
      pStmt.setInt    (2, 4);
      pStmt.setDouble (3, 4.4);

      pStmt.executeUpdate ();

      Statement stmt = con.createStatement();
      ResultSet r=stmt.executeQuery("SELECT * FROM Orders;");

      while (r.next() ) {
          System.out.println (
             r.getRow()  + "   " + r.getString(1) + "\t" +
             r.getInt(2) + "\t"  + r.getDouble(3)   );
      }

      if (con != null)   con.close();

    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

<u>**Result, AJ1015.java**</u>

```
1   green       1       1.1
2   red         2       2.2
3   blue        3       3.3
4   yellow      4       4.4
```


ResultSet METHODS TO GET COLUMNS FROM ROWS

1. The ResultSet interface has many methods to get the value from a specified column in the current row. For example:

Method Name	Return Type
getBoolean	boolean
getByte	byte
getBytes	byte[]
getDouble	double
getInt	int
getLong	long
getObject	Object
getString	String
getURL	URL

2. For most of these get methods, the column can be specified by its name (coded as a String that is not case sensitive) or by its index in the ResultSet row (the first column is 1).

   a. If multiple columns have the same name, the leftmost one is returned.

   b. Column names are designed to be used after you have used column names in your SQL command that generated the table, and may not be recognized otherwise.

3. To achieve the broadest portability, columns should be gotten left to right, and each column should be gotten only once.

4. If the ResultSet is empty, or if the cursor is before first or after last, a get method will cause an SQLException.

5. Many SQL column types can be retrieved as Strings with the getString method, but numeric values will then have to be converted to their numeric type (int, double, etc.) before they can be used in numeric operations.

6. "JDBC types" are public static final int constants that JDBC uses to identify generic SQL types. JDBC types are listed in the class java.sql.Types.

7. A reference value gotten from a ResultSet column may be null, or a numeric value may be 0 or 0.0, without causing an Exception. <u>To check for a null reference</u> use the ResultSet method r.wasNull which must be called before the next use of the ResultSet reference. <u>Use == to check basic types</u> for zero.

```
String color = r.getString(1);
    if (color == null) System.out.println("color ref null");
    if (r.wasNull())   System.out.println("color ref null");
int gallons = r.getInt(2);
    if (gallons == 0) System.out.println("gallons is zero");
```

FINALIZE METHOD

```
@Override
protected void finalize() throws Throwable {
    if (con != null) {
        con.close();
        con = null;
    }
    super.finalize();
}
```

1. Do not use the finalize method.

2. The Object class provides the callback method finalize() that may be invoked on an object when it becomes garbage (the reference count goes down to zero).

   a. A callback method is a method that gets called when an event occurs.

   b. Object's finalize method does nothing.

   c. You can override finalize() to do cleanup, such as freeing resources.

   d. The JVM never invokes the finalize method more than once for any given object.

3. The finalize() method may be called automatically by the garbage collector, but when it is called, or even if it is called, is uncertain. You can not rely on this method to be executed.

EXERCISES

NOTE:
Eclipse: to copy RoomReservation5.java from com.themisinc.u05 and FileToArray9.java from com.themisinc.u09 to com.themisinc.u10:

1. Highlight the original file
2. Right click on the highlighted file
3. In the popup, click Copy
4. Highlight com.themisinc.u10
5. Right click on the highlighted directory
6. In the popup, click Paste

1. Create E101.java in com.themisinc.u10. In the program, create a table called Corn, and load these rows of data into it.

Kernel	Can	16oz	1.79
Creamed	Pouch	15oz	1.99

   a. Select all rows and display the data on the console.
   b. Drop the table Corn.

2. READING EXERCISE, E102.java in com.themisinc.u10.

   a. Is E102.java functionally equivalent to E101.java? In what ways is it similar and different?

   b. JDBC encapsulates the interface to databases in a "vendor independent" way. How does E102.java illustrate this?

3. CASE STUDY in com.themisinc.u10

   a. Create a class called ArrayToDB10.java with the two methods described below, and any other methods you need.

   b. Create a method arrayToDB that receives the rrArray of RoomReservation5 objects. For each object in the array, the method causes a row to be inserted in a table called ReservationTable in the Java2DB.mdb database.

   c. Create a method printTable that displays all rows in ReservationTable.

   d. Copy CaseStudy9.java and call the copy CaseStudy10.java. Create an instance of ArrayToDB10 and call its method arrayToDB, passing the rrArray of RoomReservation5 objects. After the ReservationTable is populated, call ArrayToDB10's method printTable to print the data.

SOLUTIONS

E101.java in com.themisinc.u10

```
package com.themisinc.u10;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
import java.sql.ResultSet;
public class E101 {
  public static void main(String [] args) {

    String driver =
      "net.ucanaccess.jdbc.UcanaccessDriver";
    String dbURL =
      "jdbc:ucanaccess://c:/myjava/database/Java2DB.mdb";

    try {
      Class.forName (driver);
      Connection con = DriverManager.getConnection (dbURL);
      Statement stmt = con.createStatement();

      String create="CREATE TABLE Corn"
          + " (name VARCHAR(20), packaging VARCHAR(20),"
          + " weight VARCHAR(20), price DOUBLE);"  ;
      stmt.executeUpdate (create);

      stmt.executeUpdate( "INSERT INTO Corn VALUES (" +
          "'Kernel', 'Can', '16oz', '1.79');" );
      stmt.executeUpdate( "INSERT INTO Corn VALUES (" +
          "'Creamed', 'Pouch', '15oz', '1.99');" );

      ResultSet r=stmt.executeQuery("SELECT * FROM Corn;");

      while (r.next() ) {
          System.out.println (r.getRow() + ".  " +
             r.getString(1) + "\t" + r.getString(2) + "\t" +
             r.getString(3) + "\t" + r.getDouble(4) );
      }

      stmt.executeUpdate ( "DROP TABLE Corn;" );

      //r.close(); Access closes r when table Corn is dropped
      stmt.close();
      con.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Result, E101.java in com.themisinc.u10

```
1.  Kernel      Can     16oz    1.79
2.  Creamed     Pouch   15oz    1.99
```

E102.java in com.themisinc.u10

```
package com.themisinc.u10;
public class E102 {

    public static void main (String[] args) {

        String driver =
          "net.ucanaccess.jdbc.UcanaccessDriver";
        String dbURL =
          "jdbc:ucanaccess://c:/myjava/database/Java2DB.mdb";

        CarrotManager cm = new CarrotManager (driver, dbURL);

        cm.createCarrotTable();

        cm.insertRow("Sliced", 14.0, "Can", 1.89);
        cm.insertRow("Baby", 16.0, "Pouch", 2.39);

        cm.displayCarrotTable();
        cm.dropCarrotTable();
    }
}
```

CarrotManager.java in com.themisinc.u10

```
package com.themisinc.u10;
import java.sql.DriverManager;
import java.sql.Connection;
import java.sql.Statement;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
public class CarrotManager {

    private String            driver    = null;
    private String            dbURL     = null;
    private Connection        con       = null;
    private Statement         stmt      = null;
    private PreparedStatement pStmt     = null;

    private final String      sqlInsert =
        "INSERT INTO CarrotTable VALUES (?, ?, ?, ?)";

    public CarrotManager (
      String driver, String dbURL) {
        this.driver = driver;
        this.dbURL = dbURL;
    }

    public void createConnection () {
        try {
            Class.forName(driver);    //load driver
            con=DriverManager.getConnection(dbURL);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

```

    public void createCarrotTable() {
        try {
            String create="CREATE TABLE CarrotTable"
                + " (name VARCHAR(20),"
                + " ounces DOUBLE,"
                + " packaging VARCHAR(20),"
                + " price DOUBLE);"  ;
            createConnection();
            stmt = con.createStatement();
            stmt.executeUpdate (create);
            closeConnection();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public void insertRow (String name, double ounces,
    String packaging, double price) {
        try {
          createConnection();
          pStmt = con.prepareStatement (sqlInsert);

          pStmt.setString (1, name);
          pStmt.setDouble (2, ounces);
          pStmt.setString (3, packaging);
          pStmt.setDouble (4, price);

          if (pStmt.executeUpdate() == 1) { //1 is row count
              pOut ("inserted " + name);
          }
          closeConnection();
        } catch (Exception e) {
          e.printStackTrace();
        }
    }

    public void displayCarrotTable() {
        try {
            createConnection();
            stmt = con.createStatement();
            ResultSet rs = stmt.executeQuery (
                "select * from CarrotTable");

            pOut ("\nname    Ounces  Package Price");
            pOut (  "-----------------------------");

            while (rs.next()) {
                String s = rs.getString("name") + "\t"
                    + rs.getDouble("ounces")    + "\t"
                    + rs.getString("packaging") + "\t"
                    + rs.getDouble("price");
                pOut (s);
            }
```



```
            closeConnection();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public void dropCarrotTable() {
        try {
            createConnection();
            stmt = con.createStatement();
            stmt.executeUpdate ("DROP TABLE CarrotTable;");
            closeConnection();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public void closeConnection() {
        try {
            if (con != null) {
                con.close();
            }
        } catch (Exception e) {
        }
    }

    public void pOut (String s) {
        System.out.println (s);
    }
}
```

Result, E102.java in com.themisinc.u10

```
inserted Sliced
inserted Baby

name    Ounces  Package Price
-----------------------------
Sliced  14.0    Can     1.89
Baby    16.0    Pouch   2.39
```

CaseStudy10.java in com.themisinc.u10

```java
package com.themisinc.u10;
public class CaseStudy10 {

    private static String driver =
        "net.ucanaccess.jdbc.UcanaccessDriver";
    private static String dbURL =
       "jdbc:ucanaccess://c:/myjava/database/Java2DB.mdb";

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

        FileToArray9 fta = new FileToArray9 ();
        RoomReservation5[] rrArray = null;

        try {
            rrArray = fta.getArray ();
        } catch (Exception e) {
            e.printStackTrace ();
            System.exit (1);
        }

        if (rrArray == null) {
            System.err.println ("rrArray is null, exiting");
            System.exit (2);
        }

        ArrayToDB10 atdb = new ArrayToDB10 (driver, dbURL);
        try {
            atdb.arrayToDB (rrArray);
            atdb.printTable ();
            atdb.dropTable();    //needed for repeated runs
        } catch (Exception e) {
            e.printStackTrace ();
            System.exit (3);
        }
    }
}
```

ArrayToDB10.java in com.themisinc.u10

```java
package com.themisinc.u10;
import java.sql.DriverManager;
import java.sql.Connection;
import java.sql.Statement;
import java.sql.PreparedStatement;
import java.sql.ResultSet;

public class ArrayToDB10 {

    private String driver;
    private String dbURL;
    private Connection con;
    private Statement stmt;
    private PreparedStatement pStmt;

```

```
    private String sqlCreate =
        "CREATE TABLE ReservationTable"
        + " (reservationNumber INT,"
        + " seats              INT,"
        + " numberOfDays       INT,"
        + " costPerSeatPerDay  DOUBLE);"  ;

    private String sqlInsert =
        "INSERT INTO ReservationTable VALUES (?, ?, ?, ?)";

    public ArrayToDB10 (String driver, String dbURL) {
        this.driver = driver;
        this.dbURL = dbURL;
    }

    public void arrayToDB (RoomReservation5[] rrArray)
     throws Exception {
        if (rrArray == null) {
            System.err.println ("null array, exiting");
            System.exit (1);
        }
        createTable();

        int rrn;
        int seats;
        int days;
        double cost;

        for (int i=0; i<rrArray.length; i++) {
            rrn = rrArray[i].getReservationNumber();
            seats = rrArray[i].getSeats();
            days = rrArray[i].getNumberOfDays();
            cost = rrArray[i].getDayRatePerSeat();

            insertRow (rrn, seats, days, cost);
        }
    }

    private void createTable () {
        try {
          Class.forName (driver);
          con = DriverManager.getConnection (dbURL);
          stmt = con.createStatement();
          stmt.executeUpdate (sqlCreate);
          stmt.close();
          con.close();
        } catch (Exception e) {
          e.printStackTrace();
          System.exit (2);
        }
    }

```

```
    private void insertRow (
      int rrn, int seats, int days, double cost) {
        try {
          con = DriverManager.getConnection (dbURL);
          pStmt = con.prepareStatement(sqlInsert);

          pStmt.setInt    (1, rrn);
          pStmt.setInt    (2, seats);
          pStmt.setInt    (3, days);
          pStmt.setDouble (4, cost);
          pStmt.executeUpdate();

          pStmt.close();
          con.close();
        } catch (Exception e) {
          e.printStackTrace();
        }
    }

    public void dropTable () {
        try {
          con = DriverManager.getConnection (dbURL);
          stmt = con.createStatement();
          stmt.executeUpdate("DROP TABLE ReservationTable;");
          stmt.close();
          con.close();
        } catch (Exception e) {
          e.printStackTrace();
        }
    }

    public void printTable () {
       try {
            con = DriverManager.getConnection (dbURL);
            stmt = con.createStatement();
            ResultSet rs = stmt.executeQuery (
                "select * from ReservationTable");

            while (rs.next()) {
               String s =
                 rs.getInt    ("reservationNumber") + "\t"
               + rs.getInt    ("seats")             + "\t"
               + rs.getInt    ("numberOfDays")      + "\t"
               + rs.getDouble ("costPerSeatPerDay");
               System.out.println (s);
            }
            rs.close();
            stmt.close();
            con.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```


Result, CaseStudy10.java in com.themisinc.u10

```
130323    12   5    25.0
130445    14   3    35.0
130505    12   2    45.0
130614    14   1    55.0
```


UNIT 11: OPTIONAL: ADVANCED JDBC

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

1. Update a table; handle errors and warnings in properly-coded catch and finally clauses; retrieve metadata about databases and ResultSets; and use stored procedures, transactions, and batches.

UPDATE A ROW IN BOTH A ResultSet and DATABASE TABLE

1. ResultSet type and concurrency control use of the cursor and whether a ResultSet can be updated. These are specified when you create the Statement that will return the ResultSet.

   a. <u>Type specifies scrolling and sensitivity</u>:
      1) <u>Scrolling</u>: can the cursor move forward only, or forward and backward.
      2) <u>Sensitivity</u>: can the ResultSet can change due to concurrent changes in the underlying database.

   b. Types are:
      1) ResultSet.TYPE_FORWARD_ONLY, can scroll forward only, not sensitive.
      2) ResultSet.TYPE_SCROLL_INSENSITIVE, can scroll forward or backward, not sensitive.
      3) ResultSet.TYPE_SCROLL_SENSITIVE, can scroll forward or backward, and the ResultSet can change due to concurrent changes in the underlying database.

   c. <u>Concurrency makes a ResultSet readonly or updatable</u>.
      1) ResultSet.TYPE_FORWARD_ONLY
      2) ResultSet.TYPE_SCROLL_INSENSITIVE
      3) ResultSet.TYPE_SCROLL_SENSITIVE

2. To update column(s) in the current row in in the ResultSet and the database table, as shown on the facing page:

```
rs.absolute(1);              //line 30
rs.updateDouble(2, 2.00);    //line 34
rs.updateRow();              //line 35
```

   a. updateRow() cannot be called when the cursor is on the "insert row" (see paragraph 3 below). If the cursor is moved before updateRow() is called the changes are lost.

   b. cancelRowUpdates() cancels column changes in the current row, but does nothing if updateRow() has been called.

   c. deleteRow() deletes the current row from the ResultSet and the database, but cannot be called when you are on the insert row.

3. To create a new row: an updatable ResultSet has a special <u>insert row</u> where you can prepare a new row to be inserted into the ResultSet and the database table. The insertRow() method inserts the insert row's contents into the ResultSet and into the database. You must be on the insert row when you call insertRow().

```
rs.moveToInsertRow();         //move cursor to insert row
rs.updateString(1, "Tuscan");//assign column 1   "Tuscan"
rs.updateDouble(2, 1.95);     //assign column 2   1.95
rs.updateInt(3, 1264);        //assign column 3   1264
rs.insertRow();               //insert row into rs & table
rs.moveToCurrentRow();        //move cursor back to rs
```

11.03 UPDATE A ROW, EXAMPLE

AJ1103.java

```
import java.sql.*;
public class AJ1103 {
    public static void main (String[] args) throws Exception{
        String driver="net.ucanaccess.jdbc.UcanaccessDriver";
        String dbURL  =
        "jdbc:ucanaccess://c:/myjava/database/Java2DB.mdb";

        StringBuilder createKale = new StringBuilder()
            .append( "CREATE TABLE Kale "  )
            .append( "(name VARCHAR(20), " )
            .append( "price DOUBLE, "      )
            .append( "sup_id INTEGER);"    );
        System.out.println (createKale);

        Class.forName (driver);
        Connection con=DriverManager.getConnection (dbURL);
        Statement stmt = con.createStatement (
            ResultSet.TYPE_SCROLL_INSENSITIVE,   //Scrollable
            ResultSet.CONCUR_UPDATABLE );         //Updatable

        stmt.executeUpdate ( "DROP TABLE Kale;" );
        stmt.executeUpdate (createKale.toString() );
        stmt.executeUpdate( "INSERT INTO Kale VALUES ("
            + "'Curly', '1.60', '1264');"  );
        stmt.executeUpdate( "INSERT INTO Kale VALUES ("
            + "'Plain', '3.20', '1264');"  );

        ResultSet rs=stmt.executeQuery (
            "SELECT * FROM Kale WHERE name = 'Curly';" );
            rs.absolute(1);             //move cursor to row 1
            System.out.println ("Before " + rs.getString(1) +
                " " + rs.getDouble(2) + " " + rs.getInt(3) );

            rs.updateDouble(2, 2.00);
            rs.updateRow();

        ResultSet r=stmt.executeQuery("SELECT * FROM Kale;");
            while (r.next()) {System.out.println("After "
                + r.getRow() + ". " + r.getString(1) + " "
                + r.getDouble(2) + "  " + r.getInt(3) );
              }
        con.close();
    }
}
```

Result, AJ1103.java

```
CREATE TABLE Kale (name VARCHAR(20), price DOUBLE, sup_id
INTEGER);
Before Curly 1.6 1264
After 1. Plain 3.2  1264
After 2. Curly 2.0  1264
```


SQLException AND SQLWarning

1. In rare cases, one SQL statement may cause multiple database access errors. To handle this situation the SQLException class makes a <u>chain</u> of SQLException objects where each object has the information about one error.

2. <u>SQLException</u> provides:

    a. A <u>message String</u> describing the error, available via the method getMessage.

    b. An <u>SQLState String</u>, with values that follow either XOPEN or SQL:2003 conventions. The DatabaseMetaData method getSQLStateType returns the convention followed.

    c. An <u>int error code</u> that is specific to each vendor, as returned by the underlying database.

3. SQLException get methods:

    a. public String getSQLState() Returns SQLState.

    b. public int getErrorCode() Returns vendor exception code

    c. public SQLException getNextException() Returns the next chained SQLException, or null if there is none.

4. <u>SQLWarning</u> is a subclass of SQLException, and reports less severe database access problems, such as if an error occurs during a disconnection.

    a. Warnings are uncommon. The most common is DataTruncation, a subclass of SQLWarning, which warns if a database read unexpectedly truncates a value for reasons other than its having exceeded MaxFieldSize.

    b. The classes Connection, Statement, PreparedStatement, CallableStatement, and ResultSet have the getWarnings method.

    c. SQLWarnings can be chained. The SQLWarning method getNextWarning can retrieve the next object in the chain.

    d. Executing a statement automatically clears SQLWarnings from the previous statement, so warnings do not build up. You must retrieve warnings before executing your next statement.

    e. Warnings do not stop execution of an application.

catch AND finally CLAUSES

1. For an SQLException, and each object in its chain, the catch clause should retrieve:

   a. message string
   b. SQL state
   c. vendor error code

2. The finally clause in JDBC code should be used to release Connections, Statements, and ResultSets.

   a. Their resources are NOT automatically released even after your program terminates. Some of the resources may be in processes that are external to your program, in the server or database. In some cases databases or rows may remain locked, or unexpected, apparently random failures of the database may be caused.

   b. Closing your Connection releases all resources for the Connection, and Statements and ResultSets related to it.

SQLException, SQLWarning, catch, finally, EXAMPLE

<u>AJ1106.java</u>

```
import java.sql.*;
public class AJ1106 {
  public static void main (String [] args) {

    String driver = "net.ucanaccess.jdbc.UcanaccessDriver";
    String dbURL =
        "jdbc:ucanaccess://c:/myjava/database/Java2DB.mdb";
    Connection con = null;
    Statement stmt = null;
    ResultSet r = null;

    try {
        Class.forName (driver);
        con = DriverManager.getConnection (dbURL);
        stmt = con.createStatement ();
        r = stmt.executeQuery ("SELECT * FROM Kal;");

        //more JDBC code would be here

        SQLWarning w = con.getWarnings();
        while (w != null) {
            perr ("1. Message String=" + w.getMessage());
            perr ("2. SQL State=" + w.getSQLState());
            perr ("3. Error Code=" + w.getErrorCode());
            w = w.getNextWarning();
        }

    } catch (ClassNotFoundException e) {
         e.printStackTrace();

    } catch (SQLException e) {
        e.printStackTrace();
        while (e != null) {
            perr ("4. Message String=" + e.getMessage());
            perr ("5. SQL State=" + e.getSQLState());
            perr ("6. Error Code=" + e.getErrorCode());
            e = e.getNextException();
        }

    } finally {
        try {
            if (con != null) con.close();
        } catch (Exception e) {
             e.printStackTrace();
        }
    }
  }
  public static void perr (String s) {
      System.err.println (s + "\n");
  }
}
```

Result from UCanAccess, AJ1106.java

```
net.ucanaccess.jdbc.UcanaccessSQLException: UCAExc:::4.0.2 user
lacks privilege or object not found: KAL
     .
     .
     .   ---UCanAccess provided 31 lines of error trace
     .
     .
4. Message String=UCAExc:::4.0.2 user lacks privilege or object
not found: KAL

5. SQL State=42501

6. Error Code=-5501
```

Result from ODBC, AJ1106.java

```
java.sql.SQLException: [Microsoft][ODBC Microsoft Access Driver]
The Microsoft Jet database engine cannot find the input table or
query 'Kaal'.  Make sure it exists and that its name is spelled c
orrectly.
  at sun.jdbc.odbc.JdbcOdbc.createSQLException(Unknown Source)
  at sun.jdbc.odbc.JdbcOdbc.standardError(Unknown Source)
  at sun.jdbc.odbc.JdbcOdbc.SQLExecDirect(Unknown Source)
  at sun.jdbc.odbc.JdbcOdbcStatement.execute(Unknown Source)
  at sun.jdbc.odbc.JdbcOdbcStatement.executeQuery(Unknown Source)
  at AJ1106.main(AJ1106.java:14)

Error Message=[Microsoft][ODBC Microsoft Access Driver] The Micro
soft Jet database engine cannot find the input table or query 'Ka
al'.  Make sure it exists and that its name is spelled correctly.

SQL State=S0002

Error Code=-1305
```


DatabaseMetaData INTERFACE

1. A Connection's database can provide metadata describing its own tables, supported SQL grammar, stored procedures, the capabilities of this connection, etc. via a DatabaseMetaData object that can be obtained by calling the Connection method getMetaData.

2. The DatabaseMetaData interface is implemented by driver vendors to report the capabilities of the specific DBMS and specific driver working together.

    a. Different DBMSs can support different features, or implement them in different ways, or use different data types. Also, a driver may implement additional features.

    b. The term "database" in the javadoc for DatabaseMetaData refers to the driver-and-DBMS combination.

3. The DatabaseMetaData interface has more than 80 methods.

4. Not all drivers or DBMSs support updating the ResultSet. To prevent a runtime SQLException due to attempting to update a ResultSet that does not support updates, call either:

    a. boolean supportsResultSetType (int type)   Returns true if this database supports the parameter ResultSet type. The types are defined in java.sql.ResultSet. A specific ResultSet's type can be retrieved via the ResultSet's instance method getType().

    b. boolean supportsResultSetConcurrency (int type, int concurrency)   Returns true if this database supports the given concurrency type in combination with the given ResultSet type. The types are defined in java.sql.ResultSet. A specific ResultSet's concurrency can be retrieved via the ResultSet's instance method getConcurrency().

DatabaseMetaData INTERFACE, EXAMPLE

AJ1109.java

```
import java.sql.*;
public class AJ1109 {
  public static void main(String [] args) throws Exception {

    String driver = "net.ucanaccess.jdbc.UcanaccessDriver";
    String dbURL  =
        "jdbc:ucanaccess://c:/myjava/database/Java2DB.mdb";

    Class.forName(driver);
    Connection con=DriverManager.getConnection(dbURL);

    DatabaseMetaData md = con.getMetaData();

    prin ("1=" + md.getDatabaseProductName());

    prin ("2=" + md.supportsResultSetType(
        ResultSet.TYPE_SCROLL_INSENSITIVE));

    prin ("3=" + md.supportsResultSetConcurrency(
        ResultSet.TYPE_SCROLL_INSENSITIVE,
        ResultSet.CONCUR_UPDATABLE));

    con.close();
  }
  public static void prin (String s) {
    System.out.println(s);
  }
}
```

Result, AJ1109.java

```
1=UCanAccess driver for Microsoft Access databases using HSQLDB
2=true
3=true
```


ResultSetMetaData INTERFACE

1. The ResultSetMetadata interface provides methods to get data about the types and properties of ResultSet columns. Such data is useful when the SQL that generated the ResultSet is not known.

2. To work with a ResultSet when you don't have the SQL that generated the table, create a ResultSetMetaData object. Then you can use ResultSetMetaData methods to find out how many columns the ResultSet has, the characteristics of each column, whether a given column can be used in a WHERE clause, etc.

3. ResultSetMetadata has more than 20 methods. Some are:

    a. getColumnCount()
    b. getColumnName()
    c. getColumnType()
    d. getColumnDisplaySize()
    e. isSearchable()

4. ResultSetMetadata provides data similar to that obtained from reflection or introspection, which do not work on ResultSets.

ResultSetMetaData INTERFACE, EXAMPLE

<u>AJ1111.java</u>

```java
import java.sql.*;
public class AJ1111 {

    private static String driver =
        "net.ucanaccess.jdbc.UcanaccessDriver";
    private static String dbURL  =
        "jdbc:ucanaccess://c:/myjava/database/Java2DB.mdb";

    private static Statement stmt = null;
    private static ResultSet rs = null;
    private static ResultSetMetaData rsmd = null;

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

        Class.forName (driver);
        Connection con = DriverManager.getConnection (dbURL);
        stmt = con.createStatement();
        rs = stmt.executeQuery("SELECT * FROM Kale");

        printResultSet();
    }

    public static void printResultSet () throws Exception {

        rsmd = rs.getMetaData();
        int numCols = rsmd.getColumnCount();

        for (int i=1; i<=numCols; i++) {
            System.out.print(rsmd.getColumnName(i) + "\t");
        }
        System.out.println();

        while (rs.next()) {
            for (int col=1; col<=numCols; col++) {
                System.out.print (rs.getString(col) + "\t");
            }
            System.out.println();
        }
    }
}
```

<u>Result, AJ1111.java</u>

```
name    price   sup_id
Curly   1.6     1264
Plain   3.2     1264
Tuscan  2.5     1264
```


STORED PROCEDURES, callableStatement

1. A stored procedure is a group of SQL statements that form a logical unit and perform a specific task, similar to a subroutine in a programming language. For example, stored procedure operations on an employee table in a database could be: hire, promote, lookup, terminate.

    a. The stored procedure is stored in the database so it can be invoked at any later time.

    b. Each DBMS may have variations in syntax for stored procedures.

    c. JDBC has a syntax that uses curly braces, and is called an "escape syntax". When you use the escape syntax to create and call a stored procedure, your code will work without changes for most DBMSs.

        1) To call the callable procedure, enclose the call in a set of { } curly braces:

```
CallableStatement cs = con.prepareCall (
    "{ call getName (?, ?) }"
);
```

2. Stored procedures are handled via the CallableStatement interface, a subinterface of PreparedStatement.

3. Stored procedures can be compiled and executed with parameters, which are coded with more information than those for PreparedStatement.

    a. Parameter values are identified by their sequence in the list of parameters, as 1, 2, etc.

    b. Parameters must be identified as IN, OUT, or INOUT.

        1) Parameter values to be sent to the database are assigned by set methods that "register" them as IN.

        2) Parameter values to be retrieved from the database are "registered" as OUT via the registerOutParameter method; their type is specified so the driver can allocate the number of bytes for their type. Their values are retrieved via get methods after execution of the CallableStatement.

4. Three methods can be used to execute a CallableStatement.

   a. if executeQuery is used, a ResultSet can be returned.

   b. if executeUpdate is used, counts from one or more DDL statements can be returned.

   c. if execute is used, one or more counts and ResultSets can be returned.

      1) execute returns boolean. If true, the first result is a ResultSet. If false, the first result is a count or there is no result.

      2) To retrieve results, call the method getResultSet or getUpdateCount. Then call getMoreResults to move to the next result if more than one was returned.

   d. The default CallableStatement returned by the Connection method prepareCall will return default ResultSets with TYPE_FORWARD_ONLY and CONCUR_READ_ONLY. You can specify different ResultSet type and concurrency.

   e. For greatest portability, the ResultSets and counts should be processed before using get methods to get the values of OUT parameters.

5. Some drivers send the call statement to the database when prepareCall is called; others wait till the CallableStatement is executed. This may affect which method throws some SQLExceptions.

6. The DatabaseMetaData method supportsStoredProcedures reports whether a specific database supports stored procedures.

7. Advantages: callable statements are more efficient than executing SQL statements from within Java code, and are usually easy to use.

8. Disadvantages: Callable statements have vendor-specific variations in support and syntax, making them less portable.

TRANSACTIONS

1. A transaction is a group of SQL statements that work together to perform a multi-statement operation, and all statements must complete successfully before the transaction is successful. If any of the SQL statements in the transaction fail, all statements that were already applied to the database must be rolled back to ensure data integrity (aka security).

2. Why use transactions? When a multi-statement operation modifies one or more rows, if multiple users are accessing the same database they may try to access the same rows at the same time. To protect data integrity, the first transaction must complete and be committed or rolled back, before other users are allowed to access the same rows.

3. An application that uses transactions manages its Connection via the methods setAutoCommit and setTransactionIsolation.

    a. By default a Connection is auto-commit and automatically commits after executing each statement.

    b. For transaction processing:

        1) Turn off autocommit via the Connection method setAutoCommit(false);

        2) Execute all SQL statements of the transaction.

        3) If all SQL statements complete successfully, commit their changes by calling the Connection method commit, or else database changes will not be saved.

        4) If any SQL statements complete unsuccessfully, rollback all changes via the Connection method rollback.

4. A commit occurs when a statement completes.

    a. An SQL DLL statement, or SQL DML statement such as INSERT, UPDATE, or DELETE, is complete when it finishes executing.

    b. An SQL SELECT statement is complete when its ResultSet is closed.

    c. A CallableStatement, or a statement that returns multiple results, is complete when all of its ResultSets are closed and all of its update counts and output parameters have been retrieved.

TRANSACTIONS, EXAMPLE

AJ1115.java

```
import java.sql.*;
public class AJ1115 {
    public static void main (String[] args) {

        String driver="net.ucanaccess.jdbc.UcanaccessDriver";
        String dbURL =
          "jdbc:ucanaccess://c:/myjava/database/Java2DB.mdb";
        Connection con = null;
        Statement stmt = null;

        try {
            Class.forName (driver);
            con = DriverManager.getConnection (dbURL);
            stmt = con.createStatement();

            con.setAutoCommit(false);

            stmt.executeUpdate(
                "DELETE FROM Kale WHERE name = 'Curly'");
            stmt.executeUpdate( "INSERT INTO Kale VALUES " +
                "('Jersey', '2.29', '1264');"  );
            stmt.executeUpdate( "INSERT INTO Kale VALUES " +
                "('Russian', '2.50', '1264');"  );
            stmt.executeUpdate( "INSERT INTO Kale VALUES " +
                "('Tuscan', '2.89', '1264');"  );

           //con.commit();    //both commit() and rollback()
           con.rollback();    //worked in Access

        ResultSet r=stmt.executeQuery("SELECT * FROM Kale;");
           while (r.next()) {System.out.println("After "
                + r.getRow() + ". " + r.getString(1) + " "
                + r.getDouble(2) + "  " + r.getInt(3) );
           }
        } catch (Exception e) {
             e.printStackTrace();
        } finally {
            try {
                con.close();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

Result, AJ1115.java

```
After 1. Curly 1.6  1264
After 2. Plain 3.2  1265
```


TRANSACTIONS AND ISOLATION LEVELS

1. The isolation level of a transaction determines the type of locking it will have to prevent other transaction's reads.

2. Isolation level is managed by two Connection methods.

   a. setTransactionIsolation
   b. getTransactionIsolation

3. The Connection class offers the following isolation levels.

   a. TRANSACTION_NONE, transactions are not supported.

   b. TRANSACTION_READ_UNCOMMITTED, dirty reads, non-repeatable reads and phantom reads can occur.

   c. TRANSACTION_READ_COMMITTED, dirty reads are prevented; non-repeatable reads and phantom reads can occur.

   d. TRANSACTION_REPEATABLE_READ, dirty reads and non-repeatable reads are prevented; phantom reads can occur.

   e. TRANSACTION_SERIALIZABLE, dirty reads, non-repeatable reads and phantom reads are prevented.

Isolation Level	dirty reads allowed?	non-repeatable reads allowed?	phantom reads allowed?
TRANSACTION_READ_UNCOMMITTED	yes	yes	yes
TRANSACTION_READ_COMMITTED	no	yes	yes
TRANSACTION_REPEATABLE_READ	no	no	yes
TRANSACTION_SERIALIZABLE	no	no	no

1. <u>Dirty read</u> means a row changed by transactionA is read by transactionB before changes are committed. If transactionA rolls back any of its changes, transactionB will have read an invalid row. Disallowing dirty reads may lower performance due to locking overhead, and cause slower concurrency.

2. <u>Non-repeatable read</u> means transactionA reads a row, transactionB alters the row, and transactionA rereads the the row, getting different values the second time.

3. <u>Phantom read</u> means transactionA reads all rows that satisfy a WHERE condition, transactionB inserts a row that satisfies the same WHERE condition, and transactionA rereads for the same WHERE condition and retrieves different results due to the additional "phantom" row that was not present in the first read.

ACID: ATOMIC, CONSISTENT, ISOLATED, DURABLE

1. The acronym ACID stands for "Atomic, Consistent, Isolated, Durable" which are the characteristics of secure processing of database transactions.

   a. Atomic: each transaction is handled as a single unit of work, so that either all or none of its tasks are done.

   b. Consistent: the database is consistent before and after a transaction is done, because only complete, valid data is committed and partial invalid data is rolled back.

   c. Isolated: each transaction is separate. During processing of one transaction, other transactions cannot read the data in an intermediate state.

   d. Durable: committed transaction results are permanent. This usually requires database backups and transaction logs to enable restoration of committed transactions despite subsequent hardware or software failures.

TRANSACTIONS AND ISOLATION LEVELS, EXAMPLE

<u>**AJ1118.java**</u>

```
import java.sql.*;
public class AJ1118 {
    public static void main (String[] args) {

        String driver="net.ucanaccess.jdbc.UcanaccessDriver";
        String dbURL =
          "jdbc:ucanaccess://c:/myjava/database/Java2DB.mdb";
        Connection con = null;
        Statement stmt = null;

        try {
            Class.forName (driver);
            con = DriverManager.getConnection (dbURL);
            stmt = con.createStatement();
            con.setAutoCommit(false);
            con.setTransactionIsolation(
                Connection.TRANSACTION_READ_COMMITTED);
            ResultSet r =
                stmt.executeQuery("SELECT * FROM Kale;");
            while (r.next()) {System.out.println("Before "
                + r.getRow() + ". " + r.getString(1) + " "
                + r.getDouble(2) + "  " + r.getInt(3) );
            }
            stmt.executeUpdate(
                "DELETE FROM Kale WHERE name = 'Curly'");
  /*error*/ stmt.executeUpdate( "INSERT INTO Kal VALUES " +
                "('Jersey', '2.29', '1264');"  );
            con.commit();

        } catch (Exception e) {
            try {
                con.rollback();
                ResultSet r =
                    stmt.executeQuery("SELECT * FROM Kale;");
                while (r.next()) {System.out.println("After "
                    + r.getRow() + ". " + r.getString(1) +" "
                    + r.getDouble(2) + "  " + r.getInt(3) );
                }
                con.close();
            } catch (SQLException eRollback) {
            }
        }
    }
}
```

<u>**Result, AJ1118.java**</u>

```
Before 1. Curly 1.6  1264
Before 2. Plain 3.2  1265
After 1. Curly 1.6  1264
After 2. Plain 3.2  1265
```


(blank)

SQL BATCHES

1. An SQL batch is a group of SQL statements combined into one statement so they can be sent to the database at one time for efficiency.

2. Ordinary statements or prepared statements can be batched.

    a. Allowed in a batch: Statements that change the database and executeUpdate with INSERT, UPDATE, and DELETE.

    b. NOT allowed in a batch: Use of executeQuery. No statement in a batch can return a ResultSet.

3. If failure of one statement in the batch requires rollback of the entire batch, use transaction processing techniques.

4. To use a prepared statement in a batch, first bind the parameters with set methods. Next, use the PreparedStatment method addBatch to add the prepared statement to the batch.

5. The Statement method executeBatch() sends a batch of commands to the database for execution. If all commands execute successfully, it returns an int array of update counts.

    a. The order of array elements corresponds to the order commands in the batch, which corresponding to the order in which the commands were added to the batch.

    b. The element values may be:

        1) >= zero, indicates that the command was processed successfully, and this number is its update count (number of rows affected by the command's execution).

        2) Statement.SUCCESS_NO_INFO, indicates that the command was processed successfully but the number of rows affected is unknown.

        3) Statement.EXECUTE_FAILED, indicates the command failed to execute successfully, and occurs only if a driver continues to process commands after a failure. See next page for more information.

6. If a command in a batch fails, the Statement method executeBatch throws a BatchUpdateException. In this case:

   a. Your driver may or may not continue to process the remaining commands in the batch.

   b. The driver must be consistent with the particular DBMS, which means the driver must either always continue to process commands, or never continue to process commands.

   c. If the driver continues processing after a failure, you can call the BatchUpdateException method getUpdateCounts which returns an array with as many elements as commands in the batch, and at least one of the elements will be Statement.SUCCESS_NO_INFO or Statement.EXECUTE_FAILED.

   d. Implementations and return values since Java 2 allow the option of continuing to proccess commands in a batch update after a BatchUpdateException has been thrown.

7. If your program will reuse a Statement or PreparedStatement for another batch:

   a. Do not close your Connection.

   b. Call the Statement method clearBatch to empty your Statement object's current list of SQL statements before adding more.

8. If your program will reuse your PreparedStatement for another batch:

   a. Do not close your Connection.

   b. You may want to immediately release the resources used by the current parameter values by calling the PreparedStatement method clearParameters.

9. To determine whether your database and driver support batch updates, create an object of DatabaseMetaData, and call the method supportsBatchUpdates().

SQL BATCHES, EXAMPLE

AJ1122.java

```
import java.sql.*;
public class AJ1122 {
    public static void main (String[] args) {

        String driver="net.ucanaccess.jdbc.UcanaccessDriver";
        String dbURL =
          "jdbc:ucanaccess://c:/myjava/database/Java2DB.mdb";
        Connection con = null;
        PreparedStatement pStmt = null;

        String[] nameArray = {"Russian", "Tuscan"};
        int[] batchCounts;

/*1*/   try {
            Class.forName (driver);
            con = DriverManager.getConnection (dbURL);

            con.setAutoCommit (false);
            con.setTransactionIsolation(
                Connection.TRANSACTION_READ_COMMITTED);

            String sql = "DELETE FROM Kale WHERE name = ?";
            pStmt = con.prepareStatement(sql);

            for (int i=0; i<nameArray.length; i++) {
                pStmt.setString (1, nameArray[i]);
                pStmt.addBatch();
            }

        } catch (Exception e) {
            e.printStackTrace();
            try {
                con.close();
            } catch (Exception close) {
            }
        }

/*2*/   try {
            batchCounts = pStmt.executeBatch();

            System.out.println ("batch executed");
            for (int i=0; i<batchCounts.length; i++) {
                System.out.println (batchCounts[i]+" ");
            }
            con.commit();
            System.out.println ("batch committed");
```

```
        } catch (BatchUpdateException bue) {
            bue.printStackTrace();

            batchCounts = bue.getUpdateCounts();

            for (int i=0; i<batchCounts.length; i++) {
                if (batchCounts[i] ==
                Statement.EXECUTE_FAILED) {
                    System.err.println (
                    "failure in batch stmt " + i);
                }
            }
            System.err.println ("State="+bue.getSQLState());
            System.err.println ("eCode="+bue.getErrorCode());
            System.err.println ( bue.getMessage() );

            System.err.println ("\nrolling back\n");
            try {
                con.rollback();
            } catch (SQLException eRollback) { }

        } catch (SQLException e) {
            e.printStackTrace();
            try {
                System.err.println ("rollback due to " + e);
                con.rollback();
            } catch (SQLException eRollback) {
                eRollback.printStackTrace();
            }

        } finally {
            try {
                con.close();
            } catch (SQLException e) {
                 e.printStackTrace();
            }
        }
    }
}
```

Result, AJ1122.java

```
batch executed
0
0
batch committed
```


OPTIONAL: CONNECTION POOLING, CONCEPT

1. Connection pooling is the technique of creating and managing a pool of connections so they are ready for use by any application thread that needs one.

    a. Most applications need access to a JDBC connection only while processing a transaction, which takes milliseconds to complete. The dedicated connection would be idle at other times.

    b. Connection pooling enables a connection to be released (by being closed) when not in use, so it can be checked out from the pool for use by another application, thus enabling resources to be shared for more efficient use.

    c. Regardless of Exceptions or different flows-of-control, applications should close connections as soon as they finish using them, to allow resources to be recovered. Also, Statements, ResultSets, and other objects must be closed so their resources can be reused.

2. Advantages of connection pooling are: increased application performance, concurrency, and scalability; reduced time to obtain a connection; more predictable resource usage under load; ability to deploy applications on smaller or less-powerful hardware than would otherwise be required.

3. Where are resources used?

    a. Creating a database connection has the overhead of connecting over a network to the database: user authentication, and setting up transactional contexts and other aspects of the session.

    b. Each connection uses memory, CPU time, buffers, sockets, locks, context switching, etc. in the client and server.

    c. The need to manage all connection sessions for a database can be a major limiting factor in application scalability. Database resources such as locks, cursors, memory, transaction logs, statement handles, temporary tables, etc, increase with the number of concurrent connections.

4. An application may be using connection pooling transparently, without being aware of it, if it obtains connections to a data source from an application server.

5. The application server's administrator can tune the pool to maximize performance and keep applications from failing due to lack of resources.

OPTIONAL: CONNECTION POOLING, JNDI AND DataSource

1. Connection pooling is managed by a Java EE server, and is usually created via server administration tools. If an application does not use the classes and interfaces of Java EE, a standalone connection pool manager may be possible.

2. Connection pool implementations are available from JDBC driver vendors and other sources. The pool is configured in the application server by configuration files.

3. Access to the connection pool is done via JNDI, the Java Naming and Directory Interface. There are three steps:

   a. Obtain a Connection via JNDI by creating a context in which to do the lookup for your data source.

   b. Do the lookup. The name to be used in the lookup would have previously been bound to to a specific database.

   c. Use the returned reference to request your Connection.

4. The DataSource interface is implemented by a driver vendor. Three types of implementations are:

   a. Basic implementation, produces a standard Connection object identical to a connection obtained through the DriverManager.

   b. Connection pooling implementation, produces a Connection object that automatically participates in connection pooling, and works with a middle-tier connection pooling manager.

   c. Distributed transaction implementation, produces a Connection object that may be used for distributed transactions and usually participates in connection pooling. This implementation works with a middle-tier transaction manager and almost always with a connection pooling manager.

OPTIONAL: DataSource INTERFACE

1. java.sql.DriverManager is a class.

2. javax.sql.DataSource is an interface introduced in Java 1.4 that enables you use a logical name for a data source instead of a JDBC URL such as "jdbc:mysql://localhost:3306/mysql".

    a. A DataSource object is created, deployed, and managed separately from applications that use it. A driver vendor will provide a class that implements DataSource as part of its JDBC driver product.

    b. An object that implements DataSource is typically registered with a naming service based on the Java Naming and Directory Interface (JNDI) API.

    c. JNDI is typically used by application servers and web containers.

    d. A DataSource object is a factory for connections to the physical data source, and the preferred way to get a connection if JNDI and a DataSource object are available. For example, the Tomcat web container has JNDI services.

    e. A driver accessed via a DataSource object does not register itself with the DriverManager. Rather, a DataSource object is retrieved though a JNDI lookup operation and then used to create the Connection object.

3. A DataSource object has properties that are set to represent a specific data source.

    a. The properties can be modified: if the data source is moved to a new server, the server property can be changed, and any code accessing the data source need not be changed.

    b. DataSource implementations provide get and set methods for each property.

    c. The properties are initialized when the DataSource is deployed.

4. DataSource has two methods to get a connection, which are equivalent to the methods of DriverManager.

    a. Connection getConnection()

    b. Connection getConnection(String user, String pswd)

UNIT 12: AUTOBOXING, VARARGS, enum, ASSERTIONS, ANNOTATIONS

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

1. Use the following typesafety features introduced in Java 5 to help avoid errors in dealing with data types:

    autoboxing and unboxing
    varargs
    enumerations
    annotations

2. Use assertions, a typesafety feature introduced in Java 1.4.

AUTOBOXING AND UNBOXING

<u>AJ1202.java</u>

```
public class AJ1202 {
    public static void main (String[] args) {
        Double dRef1 = new Double (1.5);
        Double dRef2 = 2.5;             //autobox into dRef2

        double d = mathMethod (3.5, 4.5);//autobox args, then
                                        //unbox result to d
        System.out.println ("d=" + d);
    }
    public static Double mathMethod (Double d1, Double d2) {
        double basic1 = d1;          //unbox d1 into basic1
        double basic2 = d2;          //unbox d2 into basic2

        return basic1 + basic2;      //autobox to Double due
    }                                //to method return type
}
```

<u>Result, AJ1202.java</u>

```
d=8.0
```

===

1. <u>Autoboxing</u> is automatic conversion from a basic type variable to an object of its corresponding wrapper class. For example, if an int is used where an Integer is required, the int is autoboxed into an Integer object. Two equivalent lines:

    a. `Integer answer = 42;  //42 is autoboxed to Integer`
    b. `Integer answer = new Integer(42);`

2. <u>Unboxing</u> is automatic conversion from a wrapper class object to its corresponding basic type. For example:

    ```
    Double d  = new Double (2.5);
    double product = d * 4.0;  //d is unboxed to double
    ```

3. Autoboxing and unboxing are performed on the arguments passed to methods. If an int is passed to a method that requires an Integer, the int is autoboxed to an Integer parameter.

4. Autoboxing and unboxing enable programmers to ignore the difference between basic type variables and objects in many (but not all) situations.

VARARGS

AJ1203.java

```
1   public class AJ1203 {
2       public static void main (String[] args) {
3
4           int iRes    = calcI (1.2, 3, new Integer(4));
5           double dRes = calcD (5.6, 7, new Double(8.9));
6           prin ("i=" + iRes + ",d=" + dRes);
7
8           prin ("string", new Integer(1), 2.3, 'd', 5);
9       }
10      public static int calcI (double d, int... i) {
11          int total = (int) d;            //3, Integer(4) as 4
12          for (int param : i)
13              total = total + param;
14          return total;
15      }
16      public static double calcD (Number... n){
17          double total = 0.0;       //5.6 as Double(5.6), 7 as
18          for (Number param : n)    //Integer(7), Double(8.9)
19              total = total + param.doubleValue();
20          return total;
21      }
22      public static void prin (Object... o) {
23          for (int i=0; i<o.length; i++)
24              System.out.print (o[i] + "   ");
25      }
26  }
```

Result, AJ1203.java

```
i=8,d=21.5   string   1   2.3   d   5
```

1. Varargs, introduced in Java 5, enables you to call a method with a variable number of arguments of the same data type, which are received as an array parameter.

   a. Only one parameter, the last one, can be varargs.

   b. The varargs is coded as three period characters (also called three dots, or the ellipsis) after the data type.

2. Autoboxing with varargs allows a mixture of basic and class types to be passed to methods.

3. Because the varargs type is specified as Number on line 16 above, the method doubleValue is available on line 19.

4. Because the varargs type is specified as Object on line 22 above, any basic or class type argument(s) may be passed to the method. Basic types are autoboxed into an object of their corresponding wrapper class.

enum TYPE

1. Enumerations, called enums, were introduced in Java 5.

   a. Enums are class types. The filename of a public enum must be the same as the enum name with the .java extension.

   b. Enums implicitly extend the abstract class Enum but this should not be specified in an extends clause.

2. Your enum class definition must specify all allowed values, which are named constants called fields. Javac uses these for checking. By convention, the field names are all uppercase.

```
public enum MyColors {
    RED,
    GREEN,
    BLUE
}
```

3. An enum object must contain one of the constants defined in the enum class definition. The enum object is initialized via assignment:

```
MyColors m1;
m1 = MyColors.GREEN;
MyColors m2 = MyColors.RED;
```

4. The importance of enums is that the compiler can prevent an invalid value or type from being assigned to an enum object, but the compiler can not prevent an invalid value from being assigned to other types of variables. Uses of enums:

   a. non-boolean flags or constants that can contain one of a limited set of values.

   b. to give a name to a constant value, as an alternative to the use of final variables.

5. Comparisons of the values in enum objects can be done only via == and != operators. As of Java 1.7, a switch statement can be used (the enum constants must be specified without qualified names, such as RED, not MyColors.RED).

6. Use of a static import for an Enum, as shown on the facing page in AJ1205 on line 4, allows the use of the constant name without the Enum name qualifier on line 19, in contrast to line 16.

ENUM type, EXAMPLE

MyColors.java

```
public enum MyColors {
    RED,
    GREEN,
    BLUE
}
```

AJ1205.java

```
import java.awt.Color;
import EnumPackage.MyColors;
import static EnumPackage.MyColors.*;         //static import
public class AJ1205 {
    public static void main (String[] args) {

/*1*/   Color c = Color.ORANGE;   //compiler cannot check
        if (c != Color.RED
           && c != Color.GREEN
           && c != Color.BLUE ) {
               System.err.println ("2. bad color=" + c);
        }

/*2*/   MyColors mc1 = MyColors.RED;

/*3*/   if (mc1 != MyColors.RED) {    //without static import
           System.out.println("2. not MyColors.RED");
        }
        if (mc1 != RED) {                //with static import
           System.out.println("2. not RED");
        }

/*4*/   switch (mc1) {
            case RED: case GREEN: case BLUE:
                System.out.println ("3. " + mc1);
                break;
            default:
                System.err.println ("4. invalid");
        }

/*5*/   //MyColors mc2 = MyColors.ORANGE;
        //AJ1205.java:25: cannot find symbol
        //symbol  : variable ORANGE
        //location: class MyColors
        ///*4*/   MyColors mc2 = MyColors.ORANGE;
    }
}
```

Result, AJ1205.java

```
2. bad color=java.awt.Color[r=255,g=200,b=0]
3. RED
```

ENUM METHODS values, valueOf

MyColors.java
```
public enum MyColors {
    RED,
    GREEN,
    BLUE
}
```

AJ1206.java
```
public class AJ1206 {
    public static void main (String[] args) {

/*1*/   for (MyColors c : MyColors.values()) {//static method
            System.out.print ("1. " + c + "  ");
        }
        System.out.println ();

/*2*/   MyColors mc = MyColors.RED;
        for (MyColors ref : mc.values()) {  //instance method
            System.out.print ("2. " + ref + "  ");
        }

/*3*/   try {
            mc = MyColors.valueOf ("GREEN");  //static method
            System.out.println ("\n3. " + mc);
        } catch (IllegalArgumentException e) {
            e.printStackTrace();
        }
    }
}
```

Result, AJ1206.java
```
1. RED  1. GREEN  1. BLUE
2. RED  2. GREEN  2. BLUE
3. GREEN
```

==

1. The values method can be called as a static method qualified by the Enum name, or as an instance method qualified by a reference to an Enum object. It returns an array of all the constants that were defined for that Enum class.

2. The static method valueOf receives a parameter String and returns a reference to an Enum object that contains the corresponding Enum value or, if no such Enum value exists, throws the unchecked IllegalArgumentException. This method should be coded in a try catch.

ENUM WITH VARIABLES, METHODS, AND A CONSTRUCTOR

DyePrices.java

```
public enum DyePrices {
    ORANGE (10.60),                  //These doubles are prices.
    YELLOW (8.20),                   //These lines with constants
    TAN (7.50);                      //and prices are ctor calls.

    private double price;            //var required for the value
                                     //that goes with a constant.
    private DyePrices (double price) {      //required ctor
        this.price = price;
    }
    public void raisePrice (double p) {     //optional method
        price = price + p;
    }
    public double getPrice () {      //Method to enable a user
        return price;                //to retrieve the double
    }                                //that goes with a constant.
}
```

AJ1207.java

```
1   public class AJ1207 {
2       public static void main (String[] args) {
3           for (DyePrices dp : DyePrices.values()) {
4               dp.raisePrice(1.50);
5               System.out.print(dp+"="+dp.getPrice() + "   ");
6           }
7           DyePrices ref = DyePrices.TAN;
8           System.out.println("price of TAN=" + ref.getPrice());
9       }
10  }
```

Result, AJ1207.java

```
ORANGE=12.1   YELLOW=9.7   TAN=9.0   price of TAN=9.0
```

==

1. An enum can have a constructor, methods, and other fields (including final fields) in addition to constants.
   a. All constants must be defined before other parts.
   b. Each constant's value must be coded in ( ) parentheses.
   c. If a constructor is coded, the list of constants must end in ; semicolon.

2. If values for the constants are coded, the constructor is called when an enum variable is created, like in main line 7.
   a. The constructor parameter must have the same type as the constant values. The constructor must be private.
   b. If the constants have no values, as in MyColors, then a constructor is not needed.

ASSERTIONS

1. Assertions were introduced in Java 1.4. They are used during runtime testing and debugging.

2. An assertion states a boolean expression. If false, the JVM throws an AssertionError. If true, nothing happens.

   a. AssertionError is a subclass of Error, not Exception. Errors indicate abnormal conditions that a reasonable application should not try to catch.

   b. A method's header is not required to have a throws clause for Errors that the method might throw and not catch. The idea behind this rule is that an Error is an abnormal condition that should never occur.

3. The assert statement has two syntaxes. If an error message is specified, the AssertionError will contain it as a String and it will be printed in a stack trace.

   a. assert boolean-expression;
   b. assert boolean-expression : "error message";

4. The two statements below have the same effect, except that the assert has to be enabled on the command line when you execute the program, but the if has to be commented out or removed before you put the program into production.

   a. assert a == b : "error, a != b";
   b. if (a != b) {throw new AssertionError("error, a != b"); }

5. By default, assertions are disabled and do not affect runtime performance. To enable or disable assertions, use commandline options with the JVM. –ea and –enableassertions have the same effect. The ellipsis ... means "and all subpackages".

Option	Enable assertions for
-ea	all classes except System class
-ea:C	only class C
-ea:...	classes in the default or unnamed package and all of its subpackages
-ea:P...	classes in package P and all of its subpackages
-esa	System class (same as –enablesystemassertions)

Option	Disable assertions for
-da	all classes (same as –disableassertions)
-dsa	System class (same as –disablesystemassertions)

ASSERTIONS, EXAMPLE

<u>AJ1209.java</u>

```
public class AJ1209 {
    public static void main (String[] args) {
        int i = 1;
        int j = 2;

        assert (i < j);             //parentheses are optional
        assert i>0 : i;           //message can be a basic type
        assert i==3:"bad";         //this causes AssertionError
    }
}
```

```
$ java AJ1209                                ---UNIX commandline
$ java -ea AJ1209                            ---UNIX commandline
Exception in thread "main" java.lang.AssertionError: bad
        at AJ1209.main(AJ1209.java:8)
```

1. Options to enable and disable can be used together. To execute AJ1209 with assertions enabled for all classes in package P and its subpackages, except disabled for class C:

    ```
    $ java  -ea:P...  -da:C  AJ1209
    ```

2. In Eclipse, to enter commandline options for assertions for the JVM:

    a. Click Run, Run Configurations. Click the tab labeled "(x)=Arguments". Type your options in the "VM arguments" box. Click Run. Options may not be retained after a run.

    b. Double quotes make multiple words appear to be one. Single quotes are treated as characters in the arguments.

METADATA AND ANNOTATIONS

1. <u>Metadata</u> is "data about data" or "a different kind of data".

2. <u>Annotations</u> provide metadata about a program that is not part of the program and has no direct effect on the operation of the annotated code.

3. Annotations allow classes, interfaces, fields, and methods to be marked as having specific given attributes.

   a. Annotations allow metadata to be included in programs in a standardized way, so it is available to people reading the source code and also to the compiler, JVM, and annotation-sensitive IDEs and other tools in the development, deployment, and runtime environment.

   b. Prior to annotations, XML files were often used to hold metadata, but annotations are simpler and more robust.

   c. Possible types of information in annotations:
      1) copyright or proprietary notices; names of the programmer, maintenance, or support team, etc.; version, modification dates, and revision notes; or usage guidelines, such as for thread synchronization.
      2) deployment guidelines for application servers.

   d. Annotations are used to:
      1) suppress compiler warnings
      2) check consistency between classes, especially if a child method is intended to override a parent method.
      3) enable code to inspect other code for annotations by using reflection, and to allow software tools to generate XML or additional code such as in servlet applications.

4. Three annotation types are predefined in the Java language specification, and are located in in the java.lang package:

   @Deprecated
   @Override
   @SuppressWarnings

5. Four "meta-annotations" used in defining other annotations are in the java.lang.annotation package:

   @Documented
   @Inherited
   @Retention
   @Target

6. An annotation must immediately precede its target (the class, interface, method, or field that it applies to).

@Override IN java.lang

MyParent.java

```
1   public class MyParent {
2       private int i = 1 ;
3       public int getI () {
4           return i;
5       }
6   }
```

AJ1211.java

```
1   public class AJ1211 extends MyParent {
2       private int i = 2;
3
4       @Override
5       public int geti () {      //the name geti should be getI
6           return super.getI() + i;
7       }
8
9       public static void main (String[] args) {
10          AJ1211 ref = new AJ1211 ();
11          System.out.println("ref.getI() = " + ref.getI());
12      }
13  }
```

Result of compiling AJ1211.java on a command line

```
AJ1211.java:4: method does not override or implement a method
from a supertype
    @Override
    ^
1 error
```

Result of compiling and executing AJ1211.java without @Override

```
ref.getI() = 1
```

===

1. @Override specifies that a method overrides an inherited method, and causes a compiler error if there is no override. This error usually occurs due to spelling or capitalization errors in the overriding method name, so the inherited method would be called if the error is not corrected.

2. @Override should be coded immediately above the method to be checked by the compiler to ensure that it overrides.

@SuppressWarnings IN java.lang

1. @SuppressWarnings tells the compiler to suppress warning messages that would be caused by the immediately following "program element" (class, method, or variable).

   a. @SuppressWarnings should precede the most local (smallest) program element that it applies to.

   b. If the program element contains smaller elements with their own @SuppressWarnings annotation, then in those smaller elements both sets of warnings are suppressed.

2. @SuppressWarnings is often specified with code using legacy Collections classes.

   a. Java 5 introduced parameterized class types, called generics, in Collections, but older "raw" class types are not deprecated and are still in use.

   b. Java 5 and later compilers give a warning for each line that uses a raw Collections class. Legacy applications are slow to be updated because such changes are expensive and error-prone. If the code uses third-party libraries then the libraries have to be updated first. Hence the best solution can be @SuppressWarnings.

3. @SuppressWarnings requires one or more Strings to specify what kind of warnings to suppress. Strings recognized by different compilers may vary slightly, and unrecognized Strings are ignored. Typical Strings are:

String	Meaning
unchecked	Raw type is used instead of generic type
unused	Variable's value is not obtained by any code
deprecation	Code used is deprecated

4. When one String is specified for the @SuppressWarnings value array, the String can be coded two ways.

   a. @SuppressWarnings ("unchecked")
   b. @SuppressWarnings (value = {"unchecked"})

5. When more than one String is specified, the Strings must be coded with the identifier value and curly braces as follows.

```
@SuppressWarnings(value = {"unchecked", "deprecation"})
```

@SuppressWarnings, EXAMPLE

AJ1213.java

```
1   import java.util.ArrayList;
2   public class AJ1213 {
3
4       private static ArrayList a = new ArrayList ();
5
6       public static void main (String [] args) {
7           a.add (new String ("string") );
8           a.add (new Integer(1213) );
9           System.out.println ("ArrayList a has " + a);
10      }
11  }
```

Result of compiling AJ1213.java

```
Note: AJ1213.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Result of compiling via javac -Xlint:unchecked AJ1213.java

```
AJ1213.java:7: warning: [unchecked] unchecked call to add(E) as a
member of the raw type java.util.ArrayList
        a.add (new String ("string") );
              ^
AJ1213.java:8: warning: [unchecked] unchecked call to add(E) as a
member of the raw type java.util.ArrayList
        a.add (new Integer(1213) );
              ^
2 warnings
```

AJ1213a.java

```
1   import java.util.ArrayList;
2   public class AJ1213a {
3
4       private static ArrayList a = new ArrayList ();
5
6       @SuppressWarnings ("unchecked")
7       public static void main (String [] args) {
8           a.add (new String ("string") );
9           a.add (new Integer(1213) );
10          System.out.println ("ArrayList a has " + a);
11      }
12  }
```

Result, AJ1213a.java

```
ArrayList a has [string, 1213]
```


@Deprecated IN java.lang

<u>ClassWithDepMethod.java</u>

```
public class ClassWithDepMethod {
    /**
     * @deprecated   //javadoc tag, lowercase d
     * depMethod is deprecated because....
     * Use newMethod().
     */
    @Deprecated      //annotation, uppercase D
    public static void depMethod () {
        System.out.println ("depMethod");
    }
    public static void newMethod () {
        System.out.println ("newMethod");
    }
}
```

<u>AJ1214.java</u>

```
1   //@SuppressWarnings ("deprecation")
2   public class AJ1214 {
3       public static void main (String[] args) {
4           ClassWithDepMethod.depMethod();
5           ClassWithDepMethod.newMethod();
6       }
7   }
```

<u>Result of compiling AJ1214.java</u>

```
Note: AJ1214.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
```

<u>Result, AJ1214.java with comment // removed from line 1</u>

```
depMethod
newMethod
```

===

1. Deprecated classes, methods, constructors, and variables should be marked with the annotation @Deprecated, as well as the javadoc @deprecated tag in their javadoc comment.

    a. @Deprecated causes the word "Deprecated" to be included in javadoc documentation.
    b. @deprecated causes the word "Deprecated" and associated text to be included in javadoc documentation.

2. Code is deprecated because it is not robust and better code is available. Deprecated code is kept usually for legacy reasons. This annotation warns programmers not to make new use of deprecated code.

3. @Deprecated can be specified as @Deprecated()

OPTIONAL:
META-ANNOTATIONS @Documented @Inherited @Retention @Target

1. @Documented causes an annotation to be included in javadoc documentation as part of the public API of the targeted program elements (the default is for annotations not to be included). @Documented is useful when an annotation replaces what would otherwise be coded as a javadoc comment. @Documented should be used in annotations that could change how code operates or affect how code must be used by clients.

2. @Inherited causes an annotation type to be automatically inherited by subclasses of the class for which the annotation type has been specified.

3. @Retention specifies how long an annotation is retained. If not coded, the default is RetentionPolicy.CLASS. @Retention is specified via enum constants of RetentionPolicy:

   a. RetentionPolicy.SOURCE. Retained in the source code only, discarded by the compiler, and not put in the bytecode. Annotations on local variables can only be SOURCE.

   b. RetentionPolicy.CLASS. Retained in the bytecode but may be dropped by the JVM, hence unavailable for reflection.

   c. RetentionPolicy.RUNTIME. Retained in the bytecode and not dropped by the JVM, hence reflection methods can obtain information about the annotation. This is useful when a tool or application needs the information to determine how to deploy another application.

4. @Target specifies what program elements an annotation can be applied to. The compiler enforces this limitation. An annotation defined without @Target may be used on any program element.

   a. Program elements are specified via enum constants of ElementType:

enum constant	applies to declarations of
TYPE	Class, interface, annotation, or enum
FIELD	Field, including enum constants
METHOD	Method
PARAMETER	Parameter
CONSTRUCTOR	Constructor
LOCAL_VARIABLE	Local variable
ANNOTATION_TYPE	Annotation type
PACKAGE	Package

   b. @Target requires one or more parameters, such as

```
@Target ( {ElementType.METHOD, ElementType.FIELD} )
```


OPTIONAL: USER-DEFINED ANNOTATIONS

1. You can define your own annotations, but should minimize their number and length because they create dialects of Java and compromise its standardization and portability.

2. Annotations are defined with the @ at sign and the keyword interface. They may be public or have no access modifier. Source code for a public annotation must be in a file with the same name as the annotation and .java filename extension.

    a. Definition:
```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
@Retention (RetentionPolicy.RUNTIME)
public @interface TypeHeader {
    String developer() default "unknown";
    String[] teamMembers();
}
```

    b. Use:
```
@TypeHeader (developer = "Mimi Mee",
    teamMembers = ("Zoe", "Yoshi", "Xavier") )
```

3. The meta-annotations @Target, @Documented, @Retention, and @Inherited may be specified. If used, they must be imported.

4. The annotation:

    a. Cannot have parameters, a throws clause, or be a generic type with a type parameter.

    b. May contain "annotation element" variable declarations, specified with empty () parentheses, with either a basic data type or String, Class, Enum, or annotation data type, or may be an array of one of these types.

    c. May specify default values with the keyword default. The value null is not allowed.

    d. If there is only one element, it should be called <u>value</u>.

5. Annotations without elements are called marker annotations and are used to give an attribute to a program element.

6. When an annotation is used, each annotation element without a default value must be assigned a value. Each element may be assigned a value only one time.

7. Array elements may be assigned values as shown on lines 14 and 4 on the facing page. If the array is assigned only one value, the curly braces may be omitted: guideVersion = 5

OPTIONAL: USER-DEFINED ANNOTATION, EXAMPLE

StylePolicy.java

```
import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import java.lang.annotation.ElementType;

@Documented
@Retention (RetentionPolicy.RUNTIME)
@Target (ElementType.TYPE)

public @interface StylePolicy {               //@interface
    String style() default "project";         //default value
    int guideNumber();
    int[] guideVersion() default {10, 11, 12} ;
}
```

AJ1217.java

```
@StylePolicy (                      //Parentheses around member
    style = "training",             //names and values. Members
    guideNumber = 1217,             //with default values need
    guideVersion = {4, 5}           //not be specified but can
)                                   //be given new values.
@SuppressWarnings ("unchecked")
public class AJ1217 {
    public static void main (String[] args) {

        Class c = AJ1217.class;
        StylePolicy sp =
           (StylePolicy) c.getAnnotation (StylePolicy.class);

        p ("s="        + sp.style()
          + ", gN="    + sp.guideNumber()
          + ", gV="    + sp.guideVersion()
          + ", gV[0]=" + sp.guideVersion()[0]  );

        for (int elem : sp.guideVersion() )
            p("loop1=" + elem);
        for (int i=0; i<sp.guideVersion().length; i++)
            p("loop2=" + sp.guideVersion()[i] );
    }
    public static void p (String s) {
        System.out.println (s);
    }

```

Result, AJ1215.java

```
s=training, gN=1217, gV=[I@1c247a0, gV[0]=4
loop1=4
loop1=5
loop2=4
loop2=5
```

EXERCISES

1. The solutions download for this class has starter code for CaseStudy12.java and RoomReservation12.java. The starter code is also printed starting on page aj12.25. Put the classes in package com.themisinc.u12.

2. In CaseStudy12 in the main method, make sure that all values passed to the RoomReservation12 constructors are valid. Execute CaseStudy12 and note the output, which should be the same after you make the following changes.

3. Autoboxing.

   a. Modify the values passed by the main method to the RoomReservation12 constructor so that seats is passed as an Integer and dayRatePerSeat is passed as a Double.

   b. Do CaseStudy12 and RoomReservation12 still produce the same results? Why or why not?

4. Varargs.

   a. In RoomReservation12 create an instance variable that is a String array reference called software, and public get and set methods for it.

      1) the method setSoftware receives a String varargs parameter, and uses it to instantiate the array.
      2) the method getSoftware returns the array reference.

   b. In CaseStudy12, after creating each RoomReservation12 object, call its setSoftware method and pass a variable number of String arguments, such as:

      1) to the first object: "JDK" and "Eclipse"
      2) to the second object: "JDK", "Eclipse", and "Access"

   c. In CaseStudy12, in the loop and after the call to the method printOneReservation, create an inner loop to call the method getSoftware and print the list of software that needs to be installed for the reservation.

5. Assertion.

   a. In RoomReservation12 create a getRoomAmount method. This method needs to call calculateAmount in order to ensure that the roomAmount has been calculated. Then, the method returns the roomAmount.

b. In CaseStudy12, in the loop that calls the method printOneReservation for each object, after you print the software list, assert that the roomAount is greater than 0 and less than 9000.00. If it is not, provide the message String "out of range".

c. Enable assertions via the commandline option -ea that must be provided to the JVM. Page 12.09 has the procedure for Eclipse. After the program runs, change the upper limit on the assertion to 90.00 instead of 9000.00 to force an assertion Exception to occur.

6. Enum.

a. Create an enum called EnumCourseLength with constants ONE, TWO, THREE, FOUR, and FIVE. The constants should have associated int values 1, 2, 3, 4, and 5.

b. Modify CaseStudy12 to pass a constant of the enum instead of an int for numberOfDays to the constructor of RoomReservation12.

c. In RoomReservation12, modify the two non-null constructors to accept the enum constant. Modify the declaration of the variable numberOfDays and the methods getNumberOfDays and setNumberOfDays. In the methods calculateAmount and printOneReservation, call the get method of the enum to retrieve the int associated with the constant.

d. Does the use of an enum change the results produced by the program?

7. This exercise uses the case study classes from Unit 6 with one line changed. No solution is provided this exercise. You can do this exercise in com.themisinc.u06.

a. In RoomResWithFood6.java, add the annotation @Override immediately above the header of printOneReservation(). Execute CaseStudy6.java. The output should be the same.

b. In RoomResWithFood6.java, change the name of the method printOneReservation to printReservation.

c. Compile CaseStudy6.java again. How does the annotation in RoomResWithFood6 affect the compile? Why?

d. In RoomResWithFood6.java, change the name of the method printReservation back to printOneReservation, and leave the annotation in RoomResWithFood6.java.

SOLUTIONS

CaseStudy12.java in com.themisinc.u12

```java
package com.themisinc.u12;
public class CaseStudy12 {
    public static void main (String[] args) {

        RoomReservation12[] rrArray=new RoomReservation12[2];

        rrArray[0] = new RoomReservation12 (
            130323,
            new Integer(12),                              //3
            EnumCourseLength.FIVE,                        //6
            new Double(25.00)                             //3
        );
        rrArray[0].setSoftware ("JDK","Eclipse");         //4

        rrArray[1] = new RoomReservation12 (
            130445,
            new Integer(14),                              //3
            EnumCourseLength.THREE,                       //6
            new Double(35.00)                             //3
        );
        rrArray[1].setSoftware("JDK","Eclipse","Access"); //4

        for (RoomReservation12 elem : rrArray) {
            if (elem != null)  {
                elem.printOneReservation();

                System.out.println ("Software needed:");  //4
                for (String s : elem.getSoftware() ) {    //4
                    System.out.println ("\t" + s);        //4
                }                                         //4

                assert (elem.getRoomAmount() > 0.0        //5
                    && elem.getRoomAmount() < 90.0)       //5
                    : elem.getReservationNumber() +       //5
                      " out of range";                    //5
            }
        }
    }
}
```

EnumCourseLength.java in com.themisinc.u12

```
package com.themisinc.u12;
public enum EnumCourseLength {
    ONE   (1),
    TWO   (2),
    THREE (3),
    FOUR  (4),
    FIVE  (5);
    private int enumCourseLengthInt;

    private EnumCourseLength (int n) {
        enumCourseLengthInt = n;
    }
    public int getEnumCourseLengthInt () {
        return enumCourseLengthInt;
    }
}
```

RoomReservation12.java in com.themisinc.u12

```
package com.themisinc.u12;
import java.text.NumberFormat;
public class RoomReservation12 {

    public static final int    DEFAULT_RESERVATION_NUMBER
        = 130789;
    public static final int    DEFAULT_SEATS = 12;
    //public static final int  DEFAULT_NUMBER_OF_DAYS=5;  //6
    public static final double DEFAULT_DAY_RATE_PER_SEAT
        = 25.00;

    private int reservationNumber;
    private int seats;
    private EnumCourseLength numberOfDays;                //6
    private double dayRatePerSeat;
    private String[] software;                            //4

    private double roomAmount;

    private StringBuilder sb       = new StringBuilder();
    private StringBuilder sbMoney  = new StringBuilder();
    private StringBuilder sbInt    = new StringBuilder();

    private NumberFormat nfMoney  =
        NumberFormat.getCurrencyInstance();

    public RoomReservation12 () {
    }
    public RoomReservation12 (
      int reservationNumber,
      int seats,
      EnumCourseLength numberOfDays,                      //6
      double dayRatePerSeat) {
        setReservationNumber (reservationNumber);
        setSeats (seats);
        setNumberOfDays (numberOfDays);
```



```
      setDayRatePerSeat (dayRatePerSeat);
    }
    public RoomReservation12 (
     int reservationNumber,
     int seats,
     EnumCourseLength numberOfDays                     //6
    ) {
      this (reservationNumber, seats,
        numberOfDays, DEFAULT_DAY_RATE_PER_SEAT);
    }

    private void calculateAmount () {
      int days = numberOfDays.getEnumCourseLengthInt(); //6
      roomAmount = seats * days * dayRatePerSeat;       //6
    }

    private String formatMoney (double d) {
      sbMoney.delete (0, sbMoney.length());
      sbMoney.append (nfMoney.format(d));
      int spacesNeeded = 12 - sbMoney.length();
      for (int i=1; i<=spacesNeeded; i++) {
        sbMoney.insert(0, ' ');
      }
      return sbMoney.toString();
    }
    private String intTo12String (int param) {
      sbInt.delete (0, sbInt.length());
      sbInt.append (Integer.toString (param));
      int spacesNeeded = 12 - sbInt.length();
      for (int i=1; i<=spacesNeeded; i++) {
        sbInt.insert(0, ' ');
      }
      return sbInt.toString();
    }

    public void printOneReservation () {
      calculateAmount ();
      sb.delete (0, sb.length());
      sb.append ("\nReservation:       ");
      sb.append (   intTo12String (reservationNumber) );
      sb.append ("\nNumber of seats:   ");
      sb.append (   intTo12String (seats) );
      sb.append ("\nNumber of days:    ");
      sb.append (   intTo12String (                      //6
        numberOfDays.getEnumCourseLengthInt() ));      //6
      sb.append ("\nDay rate per seat: ");
      sb.append (   formatMoney(dayRatePerSeat));
      sb.append ("\nRoom amount:       ");
      sb.append (   formatMoney(roomAmount) + "\n");
      System.out.println (sb.toString());
    }
```

```

    public int getReservationNumber () {
        return reservationNumber;
    }
    public void setReservationNumber(int reservationNumber) {
        sb.delete (0, sb.length() );
        String s = Integer.toString (reservationNumber);
/*1*/   if (s.length() != 6) {
            sb.append ("invalid length=");
            sb.append (s.length());
            sb.append ("\n");
        }
/*2*/   if (! s.startsWith ("130") ) {
            sb.append ("does not start with 130\n");
        }
/*3*/   char c3 = s.charAt (3);
        if (c3 == s.charAt(4) && c3 == s.charAt(5) ) {
            sb.append ("chars 4, 5, and 6 are the same\n");
        }
        if (sb.length() == 0) {
            this.reservationNumber = reservationNumber;
        } else {
            sb.insert (0, "\n");
            sb.insert (0, DEFAULT_RESERVATION_NUMBER);
            sb.insert (0, " is invalid, will use ");
            sb.insert (0, reservationNumber);
            sb.insert (0, "\n");
            System.err.println (sb.toString() );
            this.reservationNumber =
                DEFAULT_RESERVATION_NUMBER;
        }
    }

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

    public EnumCourseLength getNumberOfDays () {          //6
        return numberOfDays;                              //6
    }                                                     //6
```



```
    public void setNumberOfDays (                               //6
      EnumCourseLength numberOfDays) {                          //6
        this.numberOfDays = numberOfDays;                       //6
    }                                                           //6

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

    public void setSoftware (String... software) {              //4
        this.software = software;                               //4
    }                                                           //4
    public String[] getSoftware () {                            //4
        return software;                                        //4
    }                                                           //4

    public double getRoomAmount () {                            //5
        calculateAmount ();                                     //5
        return roomAmount;                                      //5
    }                                                           //5
}
```

Result, CaseStudy12.java, with assertion upper value 90.00 executed with JVM commandline option -ea (see page 12.09 for the procedure for Eclipse)

```
Reservation:             130323
Number of seats:             12
Number of days:               5
Day rate per seat:       $25.00
Room amount:          $1,500.00

Software needed:
        JDK
        Eclipse
Exception in thread "main" java.lang.AssertionError: 130323 out
of range
        com.themisinc.u12.CaseStudy12.main(CaseStudy12.java:34)
```

Result, CaseStudy12.java, with assertion upper value 9000.00, or with upper value 90.00 but executed without JVM commandline option -ea

```
Reservation:               130323
Number of seats:               12
Number of days:                 5
Day rate per seat:         $25.00
Room amount:            $1,500.00

Software needed:
        JDK
        Eclipse

Reservation:               130445
Number of seats:               14
Number of days:                 3
Day rate per seat:         $35.00
Room amount:            $1,470.00

Software needed:
        JDK
        Eclipse
        Access
```

```java
//------------ Starter Code for CaseStudy12
package com.themisinc.u12;
public class CaseStudy12 {
    public static void main (String[] args) {
        RoomReservation12[] rrArray = new RoomReservation12[2];
        rrArray[0] = new RoomReservation12 (
                130323,
                12,
                5,
                25.00
        );
        rrArray[1] = new RoomReservation12 (
                130445,
                14,
                3,
                35.00
        );
        for (RoomReservation12 elem : rrArray) {
            if (elem != null)  {
                elem.printOneReservation();
            }
        }
    }
}
```



```
//------------ Starter Code for RoomReservation12
package com.themisinc.u12;
import java.text.NumberFormat;
public class RoomReservation12 {
    public static final int    DEFAULT_RESERVATION_NUMBER
            = 130789;
    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;
    private int reservationNumber;
    private int seats;
    private int numberOfDays;
    private double dayRatePerSeat;
    private double roomAmount;
    private StringBuilder sb      = new StringBuilder();
    private StringBuilder sbMoney = new StringBuilder();
    private StringBuilder sbInt   = new StringBuilder();
    private NumberFormat nfMoney  =
            NumberFormat.getCurrencyInstance();
    public RoomReservation12 () {
    }
    public RoomReservation12 (
            int reservationNumber, int seats,
            int numberOfDays, double dayRatePerSeat) {
        setReservationNumber (reservationNumber);
        setSeats (seats);
        setNumberOfDays (numberOfDays);
        setDayRatePerSeat (dayRatePerSeat);
    }
    public RoomReservation12 (
            int reservationNumber,
            int seats,
            int numberOfDays
    ) {
        this (reservationNumber, seats,
                numberOfDays, DEFAULT_DAY_RATE_PER_SEAT);
    }
    private void calculateAmount () {
        roomAmount = seats * numberOfDays * dayRatePerSeat;
    }
    private String formatMoney (double d) {
        sbMoney.delete (0, sbMoney.length());
        sbMoney.append (nfMoney.format(d));
        int spacesNeeded = 12 - sbMoney.length();
        for (int i=1; i<=spacesNeeded; i++) {
            sbMoney.insert(0, ' ');
        }
        return sbMoney.toString();
    }
```

```
    private String intTo12String (int param) {
        sbInt.delete (0, sbInt.length());
        sbInt.append (Integer.toString (param));
        int spacesNeeded = 12 - sbInt.length();
        for (int i=1; i<=spacesNeeded; i++) {
            sbInt.insert(0, ' ');
        }
        return sbInt.toString();
    }
    public void printOneReservation () {
        calculateAmount ();
        sb.delete (0, sb.length());
        sb.append ("\nReservation:       ");
        sb.append (   intTo12String (reservationNumber) );
        sb.append ("\nNumber of seats:   ");
        sb.append (   intTo12String (seats) );
        sb.append ("\nNumber of days:    ");
        sb.append (   intTo12String (numberOfDays) );
        sb.append ("\nDay rate per seat: ");
        sb.append (   formatMoney(dayRatePerSeat));
        sb.append ("\nRoom amount:       ");
        sb.append (   formatMoney(roomAmount) + "\n");
        System.out.println (sb.toString());
    }
    public int getReservationNumber () {
        return reservationNumber;
    }
    public void setReservationNumber(int reservationNumber) {
        sb.delete(0, sb.length());
        String s = Integer.toString (reservationNumber);
/*1*/   if (s.length() != 6) {
            sb.append ("invalid length=");
            sb.append (s.length());
            sb.append ("\n");
        }
/*2*/   if (! s.startsWith ("130") ) {
            sb.append ("does not start with 130\n");
        }
/*3*/   char c3 = s.charAt (3);
        if (c3 == s.charAt(4) && c3 == s.charAt(5) ) {
            sb.append ("chars 4, 5, and 6 are the same\n");
        }
        if (sb.length() == 0) {
            this.reservationNumber = reservationNumber;
        } else {
            sb.insert (0, "\n");
            sb.insert (0, DEFAULT_RESERVATION_NUMBER);
            sb.insert (0, " is invalid, will use ");
            sb.insert (0, reservationNumber);
            sb.insert (0, "\n");
            System.err.println (sb.toString() );
            this.reservationNumber =
                    DEFAULT_RESERVATION_NUMBER;
        }
    }
```



```
    public int getSeats () {
        return seats;
    }
    public void setSeats (int seats) {
        int assignMe = seats;
        switch (seats) {
            case 10: break;
            case 12: break;
            case 14: break;
            default: System.err.println ("Invalid seats "
                    + seats + ", will be set to "
                    + DEFAULT_SEATS);
                assignMe = DEFAULT_SEATS;
        }
        this.seats = assignMe;
    }
    public int getNumberOfDays () {
        return numberOfDays;
    }
    public void setNumberOfDays (int numberOfDays) {
        int assignMe = numberOfDays;
        if (numberOfDays < 1 || numberOfDays > 5) {
            System.err.println ("Invalid numberOfDays "
                    + numberOfDays + ", will be set to "
                    + DEFAULT_NUMBER_OF_DAYS);
            assignMe = DEFAULT_NUMBER_OF_DAYS;
        }
        this.numberOfDays = assignMe;
    }
    public double getDayRatePerSeat() {
        return dayRatePerSeat;
    }
    public void setDayRatePerSeat(double dayRatePerSeat) {
        double assignMe = dayRatePerSeat;
        if (dayRatePerSeat<25.00 || dayRatePerSeat>65.00) {
            System.err.println ("Invalid dayRatePerSeat "
                    + dayRatePerSeat + ", will be set to "
                    + DEFAULT_DAY_RATE_PER_SEAT);
            assignMe = DEFAULT_DAY_RATE_PER_SEAT;
        }
        this.dayRatePerSeat = assignMe;
    }
}
```

UNIT 13: JAVABEANS PART 1, CONVENTIONS, CODING TO AN INTERFACE, METHODS toString, equals, AND hashCode

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

1. Briefly explain JavaBeans conventions.

2. Briefly explain what interfaces are, what "coding to the interface" means, and the two new features of Java 1.8 interfaces.

3. Create classes that are JavaBeans that contain the methods toString, equals, and hashCode, and that implement the Serializable interface.

toString METHOD

1. Every class inherits the method toString() from Object, and should override it so that every object can represent its important data members as a String.

2. When System.out.println or System.out.print must print a reference to an object, they call the toString() method of the object's class.

3. An overriding method in a subclass cannot narrow the access of the overridden superclass method. toString() in Object is public, so all overriding toString() methods must be public.

4. One style of parent-child toString methods uses the literal classname and [ ] square brackets around the string of its variable values. Examples of toString output from class Child which extends class Parent are:

```
Child[valueC1,valueC2[Parent:valueP1,valueP2]]
Child[cVar1=value,cVar2=value,Parent[pVar=value]]
```

5. By hardcoding the name of the class in its toString method, the classname will always be correct. If the name of a class may change, the name can be specified in a public static final string.

6. Eclipse generates several versions of toString methods.

toString EXAMPLES

AJ1303.java

```
class A {
    private int a;
    public A (int a) {
        this.a=a;
    }
    public String toString() {
        return "A:a=" + a;
    }
}
class B extends A {
    private int b;
    public B (int a, int b) {
        super (a);
        this.b=b;
    }
    public String toString() {
        return "B:b=" + b + "[" + super.toString() + "]";
    }
}
class C extends B {
    private int c;
    public C (int a, int b, int c) {
        super(a, b);
        this.c=c;
    }
    public String toString() {
        return "C:c=" + c + "[" + super.toString() + "]";
    }
}
class D {
    private int d;
    public D (int d) {
        this.d=d;
    }
}
public class AJ1303 {
    public static void main (String[] args) {
        A a = new A (1);
        B b = new B (10, 20);
        C c = new C (100, 200, 300);
        D d = new D (99);
        System.out.println (a +"\n"+ b +"\n"+ c +"\n"+ d);
    }
}
```

Result, AJ1303.java

```
A:a=1
B:b=20[A:a=10]
C:c=300[B:b=200[A:a=100]]
D@1db9742
```


JAVABEANS CONVENTIONS

1. A JavaBean is an ordinary class that follows conventions that enable it to be used as a reusable software component by some software-building tools and frameworks such as Eclipse, Java Server Pages, Struts, and Spring.

    a. A JavaBean represents a business entity as a Java object, encapsulates its data members, and serves as a "bucket of data" to pass data over a network, etc. It may contain validation as well as any other business algorithms.

2. JavaBeans are different from Enterprise JavaBeans (EJBs).

3. JavaBeans must implement java.io.Serializable, a marker interface (it contains no methods) which gives permission for the javabean object to "persist" (have its contents stored to disk and later be read back into a program as an object).

4. JavaBeans must have a no-argument constructor (aka default constructor), and may have overloaded constructors. (If a class has no constructor the compiler provides a no-argument constructor, but because javabeans typically have overloaded constructors you must code your own no-argument constructor.)

5. JavaBeans' instance variables must be private. They are called the JavaBean's properties or attributes.

6. JavaBeans must contain get and set methods for variables whose value may be obtained or modified. A variable with no set method is called a read-only property.

    a. For a non-boolean variable called myVar, the set method must be called setMyVar() and the get method must be called getMyVar().

    b. For a boolean variable called paidUp, the set method must be called setPaidUp() and the get method must be called isPaidUp(), known as an "is method".

7. The acronym POJO means "Plain Old Java Object." The term means that a specified object is an ordinary Java Object, not an Enterprise JavaBean.

    a. There is no standard definition for POJO. They are helper classes that support your business logic.

    b. Some vendors of frameworks say POJOs do not extend other classes, but this is not generally agreed to. If POJOs do not extend, then JavaBeans can not be POJOs because JavaBeans implement Serializable and often extend other classes.

POJO JAVABEAN, EXAMPLE

Policy5.java
```
import java.io.Serializable;
public class Policy5 implements Serializable {
    private static final long serialVersionUID = 1L;

    private String policyNo;          //private instance vars
    private boolean paidUp;

    public Policy5 () {                     //no-parameter ctor
    }
    public Policy5 (String policyNo, boolean paidUp) {
        setPolicyNo (policyNo);
        setPaidUp (paidUp);
    }

    public String getPolicyNo () {
        return policyNo;
    }
    public void setPolicyNo (String policyNo) {
       this.policyNo = policyNo;
    }
    public boolean isPaidUp () {
        return paidUp;
    }
    public void setPaidUp (boolean paidUp) {
        this.paidUp = paidUp;
    }
    @Override
    public String toString () {
        return "Policy5:" + policyNo + "," + paidUp;
    }
}
```

AJ1305.java
```
public class AJ1305 {
    public static void main (String[] args) {

        Policy5 p1 = new Policy5 ();
            p1.setPolicyNo ("WL");
            p1.setPaidUp (false);
        Policy5 p2 = new Policy5 ("AD", true);

        System.out.print (p1.getPolicyNo() + " is " +
            (p1.isPaidUp() ? "" : "not ") + "paid up, ");
        System.out.println (p2.getPolicyNo() + " is " +
            (p2.isPaidUp() ? "" : "not ") + "paid up");
    }
}
```

Result, AJ1305.java
```
WL is not paid up, AD is paid up
```

JAVABEANS AND "CODING TO THE INTERFACE"

PolicyInterface.java

```
public interface PolicyInterface {
    String  getPolicyNo ();                         //methods are
    void    setPolicyNo (String policyNo);          //implicitly
    boolean isPaidUp    ();                         //public and
    void    setPaidUp   (boolean paidUp);           //abstract
    String  toString    ();
}
```

Policy7.java

```
import java.io.Serializable;
public class Policy7
  implements PolicyInterface, Serializable {
    private static final long serialVersionUID = 1L;

    private String policyNo;
    private boolean paidUp;

    public Policy7 () {
     }
    public Policy7 (String policyNo, boolean paidUp) {
        setPolicyNo (policyNo);
        setPaidUp (paidUp);
    }

    @Override
    public String getPolicyNo () {
        return policyNo;
    }
    @Override
    public void setPolicyNo (String policyNo) {
       this.policyNo = policyNo;
    }
    @Override
    public boolean isPaidUp () {
        return paidUp;
    }
    @Override
    public void setPaidUp (boolean paidUp) {
        this.paidUp = paidUp;
    }
    @Override
    public String toString () {
        return "Policy7:" + policyNo + "," + paidUp;
    }
}
```

CODING TO THE INTERFACE, EXAMPLE

AJ1307.java

```
public class AJ1307 {
    public static void main (String[] args) {

        PolicyInterface p1 = new Policy7 (); //interface ref
        p1.setPolicyNo ("WL");
        p1.setPaidUp (false);

        PolicyInterface p2 = new Policy7 ("AD", true);

        System.out.println (p1 + "   " + p2);
    }
}
```

Result, AJ1307.java

```
Policy7:WL,false   Policy7:AD,true
```

1. An interface that specifies all methods required in a POJO or JavaBean enables the compiler to verify that the methods in the POJO or JavaBean are correct. Such interfaces are used in early planning and coding a new project, and define the "role played" by the class in the project, and enable developers to "code to the interface".

2. @Override is satisfied if a method:
    a. implements a method required by an interface, or
    b. overrides an inherited method such as toString(), or a method required by an abstract ancestor class.

3. The class or interface type of an object's reference determines which identifiers are in scope in the object.

    a. An interface type reference can be used only to access identifiers in the interface. You can use instanceof and cast the reference to another class or interface type.

    b. Even if the method toString is not in the interface it can be called via Runtime Polymorphism because it overrides the toString method inherited from Object which is the ancestor of all classes. (Java acts as if Object is the ancestor of both interfaces and classes.)

    c. It is considered good style to put all methods that will need to be called in the interface, including those inherited from Object and overridden, such as toString.

JAVA 1.8 INTERFACES

1. Starting in Java 1.8 interfaces may contain default methods which consist of concrete methods that implementing classes can use "as is" or override.

   a. Default methods in interfaces reduce the need for "adapter classes" that provide method stubs (concrete methods with no statements inside their curly braces) to enable classes to implement interfaces that require methods unneeded by the implementing class.

   b. Default methods support lambda expressions, introduced in Java 1.8.

2. Default methods can be static. Only the keyword "static" is specified, not the keyword "default". To be used in an implementing class, the static method name must be qualified by the interface name.

   a. Static methods cannot be overridden in an interface or class.

   b. In Java 1.8 the interface java.util.Comparator, covered later in this course, has static methods reverseOrder() for descending sorts, and nullsLast() to support null objects.

3. Existing interfaces can be altered by adding default and/or static methods.

PurchaseInterface.java

```
public interface PurchaseInterface {
    double  getPrice ();                     //methods implicitly
    void    setPrice (double price);         //public abstract
    String  toString ();

    default String getShipper () {           //implicitly public
        return "USPS";
    }

    static String getShipperOptions () { //implicitly public
        return new String ("USPS, UPS, FEDEX") ;
    }
}
```


JAVA 1.8 INTERFACE EXAMPLE

Purchase.java

```
import java.io.Serializable;
public class Purchase
  implements PurchaseInterface, Serializable {
    private static final long serialVersionUID = 1L;

    private double price;

    public Purchase () {
    }
    public Purchase (double price) {
        setPrice (price);
    }

    @Override
    public double getPrice () {
        return price;
    }
    @Override
    public void setPrice (double price) {
       this.price = price;
    }
    @Override
    public String toString () {
        return "Purchase:price=" + price +
        ",shipper=" + getShipper() +
        ",options=" + PurchaseInterface.getShipperOptions();
    }
}
```

AJ1309.java

```
public class AJ1309 {
    public static void main (String[] args) {

        PurchaseInterface p = new Purchase(); //interface ref
        p.setPrice (123.45);

        System.out.println (p);
    }
}
```

Result, AJ1309.java

```
Purchase:price=123.45,shipper=USPS,options=USPS, UPS, FEDEX
```


equals METHOD

<u>Policy11.java</u>

```
import java.io.Serializable;
public class Policy11
  implements PolicyInterface, Serializable {

    private String policyNo;
    private boolean paidUp;

    public Policy11 () {
    }
    public Policy11 (String policyNo, boolean paidUp) {
        setPolicyNo (policyNo);
        setPaidUp (paidUp);
    }

    @Override
    public String getPolicyNo () {
        return policyNo;
    }
    @Override
    public void setPolicyNo (String policyNo) {
       this.policyNo = policyNo;
    }
    @Override
    public boolean isPaidUp () {
        return paidUp;
    }
    @Override
    public void setPaidUp (boolean paidUp) {
        this.paidUp = paidUp;
    }
    @Override         //overrides PolicyInterface and Object
    public String toString () {
        return "Policy11:" + policyNo + "," + paidUp;
    }
    @Override                                   //overrides Object
    public boolean equals (Object o) {
        if (this == o) return true;           //same object?
        if (o == null) return false;    //is parameter null?

        if (o instanceof Policy11) {      //same class type?
            Policy11 p = (Policy11) o;    //make ref of type
            if (policyNo != null
               && policyNo.equals(p.getPolicyNo() )
               && paidUp == p.isPaidUp() ) {
                return true;
            }
        }
        return false;
    }
}
```

AJ1311.java

```
public class AJ1311 {
    public static void main (String[] args) {

        PolicyInterface p1 = new Policy11 ("WL", false);
        PolicyInterface p2 = new Policy11 ("WL", false);
        PolicyInterface p3 = new Policy11 ("WL", true);
        PolicyInterface p4 = new Policy11 ("TL", false);

        p ("p1 == p1: "      + (p1 == p1)   );
        p ("p1.equals(p1): " + p1.equals(p1));

        p ("p1.equals(p2): " + p1.equals(p2));
        p ("p1.equals(p3): " + p1.equals(p3));
        p ("p1.equals(p4): " + p1.equals(p4));

    }
    public static void p (String s) {
        System.out.println (s);
    }
}
```

Result, AJ1311.java

```
p1 == p1: true
p1.equals(p1): true
p1.equals(p2): true
p1.equals(p3): false
p1.equals(p4): false
```

==

1. Every class in Java inherits an equals method from Object.

    a. The equals method in Object compares this reference to the parameter reference via == which compares the references only, and returns true if both references point to the same object.

    b. An overriding method cannot narrow the access of the overridden method. The equals method in Object is public, so methods that override it must be public.

2. Every class, including JavaBeans, should override the equals method of Object so an object can compare itself to another object and determine whether the other object:

    a. has the same class type, and
    b. has same values in its important data members.

3. Eclipse can generate equals methods.

SUBCLASS toString AND equals METHODS

TPolicy13.java

```
import java.io.Serializable;
public class TPolicy13 extends Policy11
  implements PolicyInterface, Serializable {

    private String termType;

    public TPolicy13 () {
    }
    public TPolicy13 (
    String policyNo, boolean paidUp, String termType) {
        super (policyNo, paidUp);
        setTermType (termType);
    }

    public String getTermType () {
        return termType;
    }
    public void setTermType (String termType) {
       this.termType = termType;
    }

    @Override
    public boolean equals (Object o) {

        if (o == null) return false;
        if (this == o) return true;

        //inherited variables must have the same values
        if (! super.equals(o) ) return false;

        //if they are the same class, the same Class object
        //is returned
        if (this.getClass() != o.getClass()) return false;

        TPolicy13 t = (TPolicy13) o;

        if (termType == null  &&  t.getTermType() != null) {
            return false;
        }

        if ( (termType != null) &&
             (! termType.equals (t.termType)) ) {
            return false;
        }

        return true;
    }

```

```
    @Override
    public String toString () {
        return "TPolicy13:" + termType +
            "[" + super.toString() + "]";
    }
}
```

AJ1313.java

```
public class AJ1313 {
    public static void main (String[] args) {

        PolicyInterface p1 = new TPolicy13("TL", true, "A");
        PolicyInterface p2 = new TPolicy13("TL", true, "A");
        PolicyInterface p3 = new TPolicy13("TL", false,"A");
        PolicyInterface p4 = new TPolicy13("TL", true, "B");

        p ("p1 == p1: "      + (p1 == p1)   );
        p ("p1.equals(p1): " + p1.equals(p1));

        p ("p1.equals(p2): " + p1.equals(p2));
        p ("p1.equals(p3): " + p1.equals(p3));
        p ("p1.equals(p4): " + p1.equals(p4));

        p ("\n" + p1);
    }
    public static void p (String s) {
        System.out.println (s);
    }
}
```

Result, AJ1313.java

```
p1 == p1: true
p1.equals(p1): true
p1.equals(p2): true
p1.equals(p3): false
p1.equals(p4): false

TPolicy13:A[Policy11:TL,true]
```


CREATING YOUR hashCode METHOD

1. Every class inherits a hashCode method from java.lang.Object.

2. The hashCode method of Object should be overridden by every class, including JavaBeans, so that an object of the class can represent its important data members as a hashcode.

   a. If you are not trained in hashing algorithms, you can create a useable hashcode by using the resources of the String class and the wrapper classes (except Integer, which only returns the int).

      1) Wrapper classes that wrap the basic data types have hashCode methods that use the wrapped value.

      2) The String class hashCode method creates a hash code from String objects. You can call your class's toString method, obtain the returned String, and then call the returned String's hashcode method.

   b. Some people recommend multiplying the return value of the String or Wrapper class hashCode methods by a prime number such as 11, 13, or 17.

3. Eclipse can generate hashCode methods.

OPTIONAL

4. The Collection and Map interfaces have methods that match parameters passed as Objects to elements, keys, or values.

   a. These methods include get, containsKey, containsValue, contains, and remove. These methods are implemented by several classes, such as Hashtable, HashSet, and HashMap.

   b. Classes used as elements or values should override the equals method because this method is used to handle elements and values.

   c. Classes used as keys should override both the equals and hashCode methods because these two methods are used to handle keys.

hashCode EXAMPLE

AJ1315.java

```
public class AJ1315 {
    public static void main (String[] args) {

        Data15 ref = new Data15 (1.5, "two");
        System.out.println ("returned=" + ref.hashCode());
    }
}
```

Data15.java

```
public class Data15 {
    private double d;
    private String s;
    public Data15 (double d, String s) {
        this.d = d;
        this.s = s;
    }
    @Override
    public int hashCode() {
        Double num = new Double(d * 11.11); //create Double
        int hashD = num.hashCode();         //so you can use
                                            //its hashCode()

        int hashS = s.hashCode();    //use String hashCode()

        System.out.println ("D="+hashD + "," + "S="+hashS);
        return hashD + hashS;
    }
}
```

Result, AJ1315.java

```
D=814972215,S=115276
returned=815087491
```

==

1. The equals and hashCode methods of a class should use the same important variables so their output is consistent.

2. The hashCode method must return the same int during one execution of a program if the method is called multiple times on the same object while its important variables contain the same data.

3. If the equals method finds two objects to be equal, the hashCode method should return the same int for them.
   If the equals method finds two objects to be unequal the hashCode method should return different ints for them.

SUBCLASS hashcode METHOD

<u>**Policy11.java with hashCode method added at the end**</u>

```
import java.io.Serializable;
public class Policy11
  implements PolicyInterface, Serializable {

    private String policyNo;
    private boolean paidUp;

~~

    @Override                             //overrides Object
    public int hashCode () {
        String s = toString();
        return s.hashCode();
    }
}
```

<u>**TPolicy13.java with hashCode method added at the end**</u>

```
import java.io.Serializable;
public class TPolicy13 extends Policy11
  implements PolicyInterface, Serializable {

    private String termType;//the class has only one instance
                            //var which is a String
~~

    @Override                             //overrides Policy9
    public int hashCode () {
        int superHashCode = super.hashCode();
        int thisHashCode  = termType.hashCode();

        return thisHashCode + superHashCode;
    }
}
```

<u>**AJ1316.java**</u>

```
public class AJ1316 {
    public static void main (String[] args) {

        Policy11 p1  = new Policy11 ("WL", false);
        TPolicy13 p2 = new TPolicy13 ("TL", true, "A");
        System.out.println ("1=" + p1.hashCode() + ", "
            + "2=" + p2.hashCode());
    }
}
```

<u>**Result, AJ1316.java**</u>

```
1=963515519, 2=309162621
```

EXERCISES

1. The solutions download for this class has starter code for RoomReservation13.java and RoomResWithFood.java. The starter code is also printed starting on page aj13.27. Put the classes for this exercise in package com.themisinc.u13.

2. In RoomReservation13.java and RoomResWithFood13.java, add the three additional methods toString, equals, and hashCode.

3. In CaseStudy13 in the main method:

   a. In the loop that calls the printOneReservation method, also call the methods toString and hashCode, and print the results of the calls to these methods.

   b. After the loop, call the equals method of the first RoomReservation13 object three times as described below. Print each result.

      1) For the first call, pass the reference to the same object as the parameter.

      2) For the second call, pass the reference to the other RoomReservation13 object as the parameter.

      3) For the third call, pass the reference to one of the RoomResWithFood13 objects as the parameter.

   c. Next, call the equals method of the first RoomResWithFood13 object three times as described below. Print each result.

      1) For the first call, pass the reference to the same object as the parameter.

      2) For the second call, pass the reference to the other RoomResWithFood13 object as the parameter.

      3) For the third call, pass the reference to one of the RoomReservation13 objects as the parameter.

OPTIONAL READING EXERCISE

Read the interface and classes on page aj13.26.

a. Why is it necessary to create a reference of type Breed?
b. Where does runtime polymorphism occur on line 48?

SOLUTIONS

<u>CaseStudy13.java in com.themisinc.u13</u>

```
package com.themisinc.u13;
public class CaseStudy13 {
    public static void main (String[] args) {

        RoomReservation13[] rrArray = {

            new RoomReservation13 (
                130323, 12, 5, 25.00),
            new RoomReservation13 (
                130445, 14, 3),

            new RoomResWithFood13 (
                130505, 12, 5, 45.00, true, true
                ),
            new RoomResWithFood13 (
                130614, 14, 3, true, false
                ),
        };

        for (RoomReservation13 elem : rrArray) {
            if (elem != null)  {
                elem.printOneReservation();
                System.out.println (
                    "toString="   + elem +
                    "\nhashCode=" + elem.hashCode()
                );
            }
        }

        System.out.println (
          "\n0 to 0=" + rrArray[0].equals(rrArray[0]) +
          ", 0 to 1=" + rrArray[0].equals(rrArray[1]) +
          ", 0 to 2=" + rrArray[0].equals(rrArray[2]) +

          "\n2 to 2=" + rrArray[2].equals(rrArray[2]) +
          ", 2 to 1=" + rrArray[2].equals(rrArray[1]) +
          ", 2 to 0=" + rrArray[2].equals(rrArray[0])
        );
    }
}
```


RoomReservation13.java in com.themisinc.u13

```
package com.themisinc.u13;
import java.text.NumberFormat;

public class RoomReservation13 {

    public static final int    DEFAULT_RESERVATION_NUMBER
        = 130789;
    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;

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

    private double roomAmount;

    private StringBuilder sb      = new StringBuilder();
    private StringBuilder sbMoney = new StringBuilder();
    private StringBuilder sbInt   = new StringBuilder();

    private NumberFormat nfMoney  =
        NumberFormat.getCurrencyInstance();

    public RoomReservation13 () {
    }
    public RoomReservation13 (
      int reservationNumber, int seats,
      int numberOfDays, double dayRatePerSeat) {
        setReservationNumber (reservationNumber);
        setSeats (seats);
        setNumberOfDays (numberOfDays);
        setDayRatePerSeat (dayRatePerSeat);
    }
    public RoomReservation13 (
      int reservationNumber,
      int seats,
      int numberOfDays
    ) {
        this (reservationNumber, seats,
            numberOfDays, DEFAULT_DAY_RATE_PER_SEAT);
    }

    private void calculateAmount () {
        roomAmount = seats * numberOfDays * dayRatePerSeat;
    }

```



```
    public String formatMoney (double d) {
        sbMoney.delete (0, sbMoney.length());
        sbMoney.append (nfMoney.format(d));
        int spacesNeeded = 12 - sbMoney.length();
        for (int i=1; i<=spacesNeeded; i++) {
            sbMoney.insert(0, ' ');
        }
        return sbMoney.toString();
    }
    private String intTo12String (int param) {
        sbInt.delete (0, sbInt.length());
        sbInt.append (Integer.toString (param));
        int spacesNeeded = 12 - sbInt.length();
        for (int i=1; i<=spacesNeeded; i++) {
            sbInt.insert(0, ' ');
        }
        return sbInt.toString();
    }

    public void printOneReservation () {
        calculateAmount ();
        sb.delete (0, sb.length());
        sb.append ("\nReservation:       ");
        sb.append (   intTo12String (reservationNumber) );
        sb.append ("\nNumber of seats:   ");
        sb.append (   intTo12String (seats) );
        sb.append ("\nNumber of days:    ");
        sb.append (   intTo12String (numberOfDays) );
        sb.append ("\nDay rate per seat: ");
        sb.append (   formatMoney(dayRatePerSeat));
        sb.append ("\nRoom amount:       ");
        sb.append (   formatMoney(roomAmount) + "\n");
        System.out.println (sb.toString());
    }

    public int getReservationNumber () {
        return reservationNumber;
    }
    public void setReservationNumber(int reservationNumber) {
        sb.delete(0, sb.length());
        String s = Integer.toString (reservationNumber);
/*1*/   if (s.length() != 6) {
            sb.append ("invalid length=");
            sb.append (s.length());
            sb.append ("\n");
        }
/*2*/   if (! s.startsWith ("130") ) {
            sb.append ("does not start with 130\n");
        }
/*3*/   char c3 = s.charAt (3);
        if (c3 == s.charAt(4) && c3 == s.charAt(5) ) {
            sb.append ("chars 4, 5, and 6 are the same\n");
        }
        if (sb.length() == 0) {
            this.reservationNumber = reservationNumber;
```



```
        } else {
            sb.insert (0, "\n");
            sb.insert (0, DEFAULT_RESERVATION_NUMBER);
            sb.insert (0, " is invalid, will use ");
            sb.insert (0, reservationNumber);
            sb.insert (0, "\n");
            System.err.println (sb.toString() );
            this.reservationNumber =
                DEFAULT_RESERVATION_NUMBER;
        }
    }

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

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

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



```

    @Override
    public String toString() {
        StringBuilder builder = new StringBuilder();
        builder.append("RoomReservation13[")
            .append("reservationNumber=")
            .append(reservationNumber)
            .append(",seats=").append(seats)
            .append(",numberOfDays=").append(numberOfDays)
            .append(",dayRatePerSeat=").append(dayRatePerSeat)
            .append("]");
        return builder.toString();
    }

    @Override
    public int hashCode() {
        String s = toString();
        int hash = s.hashCode();
        return hash * 31;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null) return false;
        if (getClass() != obj.getClass()) return false;

        RoomReservation13 other = (RoomReservation13) obj;

        if (reservationNumber != other.reservationNumber
        ||  seats             != other.seats
        ||  numberOfDays      != other.numberOfDays
        ||  dayRatePerSeat    != other.dayRatePerSeat)
            return false;

        return true;
    }
}
```

RoomResWithFood13.java in com.themisinc.u13

```
package com.themisinc.u13;
public class RoomResWithFood13 extends RoomReservation13 {

    public static final double AM_COST_PER_PERSON=9.00;
    public static final double PM_COST_PER_PERSON=8.00;

    private boolean amService;
    private boolean pmService;

    private double foodAmount;
    private StringBuilder sBldr = new StringBuilder();

```

```
    public RoomResWithFood13 () {
    }

    public RoomResWithFood13(
      int reservationNumber, int seats,
      int numberOfDays, double dayRatePerSeat,
      boolean amService, boolean pmService
    ) {
        super (reservationNumber, seats,
             numberOfDays, dayRatePerSeat);
        setAmService (amService);
        setPmService (pmService);
    }

    public RoomResWithFood13(
      int reservationNumber, int seats,
      int numberOfDays,
      boolean amService, boolean pmService
    ) {
        this (reservationNumber, seats,
        numberOfDays,
        RoomReservation13.DEFAULT_DAY_RATE_PER_SEAT,
        amService, pmService);
    }

    private void calculateAmount () {
        double perDay = 0.0;
        if (isAmService() ) {
            perDay = AM_COST_PER_PERSON;
        }
        if (isPmService() ) {
            perDay = perDay + PM_COST_PER_PERSON;
        }
        foodAmount = getSeats() * getNumberOfDays() * perDay;
    }

    public void printOneReservation () {
        calculateAmount();
        super.printOneReservation();
        sBldr.delete (0, sBldr.length() );
        sBldr.append ("**Food Charges:");
        sBldr.append ("\n  Food:            ");
        sBldr.append (formatMoney(foodAmount) );
        sBldr.append ("\n");
        System.out.println (sBldr.toString() );
    }

    public boolean isAmService () {
        return amService;
    }
    public void setAmService (boolean amService) {
        this.amService = amService;
    }
```



```java

    public boolean isPmService () {
        return pmService;
    }
    public void setPmService (boolean pmService) {
        this.pmService = pmService;
    }

    @Override
    public String toString() {
        StringBuilder builder = new StringBuilder();
        builder.append("RoomResWithFood13[")
            .append("amService=").append(amService)
            .append(",pmService=").append(pmService)
            .append(",[").append(super.toString() )
            .append("]]");
        return builder.toString();
    }

    @Override
    public int hashCode() {
        int superHash = super.hashCode();
        String s = toString();
        int hash = s.hashCode();
        return (hash * 17) + superHash;
    }

   @Override
    public boolean equals(Object obj) {

        if (this == obj)                    return true;
        if (obj == null)                    return false;
        if (getClass() != obj.getClass()) return false;
        if (! super.equals(obj) )          return false;

        RoomResWithFood13 rrwf = (RoomResWithFood13) obj;

        if (amService != rrwf.isAmService()
        ||  pmService != rrwf.isPmService() )
            return false;

        return true;
    }
}
```

Result, CaseStudy13.java in com.themisinc.u13

```
Reservation:              130323
Number of seats:              12
Number of days:                5
Day rate per seat:        $25.00
Room amount:           $1,500.00

toString=RoomReservation13[reservationNumber=130323,seats=12,numb
erOfDays=5,dayRatePerSeat=25.0]
hashCode=-1259867672

Reservation:              130445
Number of seats:              14
Number of days:                3
Day rate per seat:        $25.00
Room amount:           $1,050.00

toString=RoomReservation13[reservationNumber=130445,seats=14,numb
erOfDays=3,dayRatePerSeat=25.0]
hashCode=1094525669

Reservation:              130505
Number of seats:              12
Number of days:                5
Day rate per seat:        $45.00
Room amount:           $2,700.00

**Food Charges:
  Food:                $1,020.00

toString=RoomResWithFood13[amService=true,pmService=true,[RoomRes
ervation13[reservationNumber=130505,seats=12,numberOfDays=5,dayRa
tePerSeat=45.0]]]
hashCode=1157970048

Reservation:              130614
Number of seats:              14
Number of days:                3
Day rate per seat:        $25.00
Room amount:           $1,050.00

**Food Charges:
  Food:                  $378.00

toString=RoomResWithFood13[amService=true,pmService=false,[RoomRe
servation13[reservationNumber=130614,seats=14,numberOfDays=3,dayR
atePerSeat=25.0]]]
hashCode=-930296640

0 to 0=true, 0 to 1=false, 0 to 2=false
2 to 2=true, 2 to 1=false, 2 to 0=false
```



```
package com.themisinc.u13;

interface Breed {                   //Implementing classes must
    String getBreed() ;             //code one method, getBreed()
}

abstract class Pet {                //Subclasses must code one
    private String name;            //method, getFavorite(), and
    public Pet (String n) {         //inherit name and getName()
        name=n;
    }
    public String getName() {return name;}
    abstract String getFavorite();
}

class Cat extends Pet implements Breed {  //Must code methods
    private String favoritePerch;         //getFavorite() and
    public Cat (String n, String f) {     //getBreed()
        super(n);                         //Will inherit name
        favoritePerch = f;                //and getName()
    }
    public String getFavorite() {return favoritePerch;}
    public String getBreed() {return "Tabby Cat";}
}
class Gerbil extends Pet implements Breed {
    private String favoriteToy;
    public Gerbil (String n, String f) {
        super(n);
        favoriteToy = f;
    }
    public String getFavorite() {return favoriteToy;}
    public String getBreed() {return "Mongolian Gerbil";}
}

public class E131 {
  public static void main (String[] args) {

    Pet[] a = {
        new Cat ("Fluffy", "window sill"),
        new Gerbil ("Gerbert", "running wheel")
    };

    for (Pet p : a) {               //Pet ref p has identifiers
                                    //getName() and getFavorite()
       Breed b = (Breed) p;         //Breed ref b has identifier
                                    //getBreed()
       System.out.println (p.getName() + ", " +
       b.getBreed() + ", likes the "+ p.getFavorite() );
    }
  }
}
```

Result, E131.java

```
Fluffy, Tabby Cat, likes the window sill
Gerbert, Mongolian Gerbil, likes the running wheel
```

```
// Starter code for RoomReservation13.java --------------------
package com.themisinc.u13;
import java.text.NumberFormat;

public class RoomReservation13 {

    public static final int    DEFAULT_RESERVATION_NUMBER
        = 130789;
    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;

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

    private double roomAmount;

    private StringBuilder sb      = new StringBuilder();
    private StringBuilder sbMoney = new StringBuilder();
    private StringBuilder sbInt   = new StringBuilder();

    private NumberFormat nfMoney  =
        NumberFormat.getCurrencyInstance();

    public RoomReservation13 () {
    }
    public RoomReservation13 (
      int reservationNumber, int seats,
      int numberOfDays, double dayRatePerSeat) {
        setReservationNumber (reservationNumber);
        setSeats (seats);
        setNumberOfDays (numberOfDays);
        setDayRatePerSeat (dayRatePerSeat);
    }
    public RoomReservation13 (
      int reservationNumber,
      int seats,
      int numberOfDays
    ) {
        this (reservationNumber, seats,
            numberOfDays, DEFAULT_DAY_RATE_PER_SEAT);
    }

    private void calculateAmount () {
        roomAmount = seats * numberOfDays * dayRatePerSeat;
    }

    public String formatMoney (double d) {
        sbMoney.delete (0, sbMoney.length());
        sbMoney.append (nfMoney.format(d));
        int spacesNeeded = 12 - sbMoney.length();
```



```
        for (int i=1; i<=spacesNeeded; i++) {
            sbMoney.insert(0, ' ');
        }
        return sbMoney.toString();
    }
    private String intTo12String (int param) {
        sbInt.delete (0, sbInt.length());
        sbInt.append (Integer.toString (param));
        int spacesNeeded = 12 - sbInt.length();
        for (int i=1; i<=spacesNeeded; i++) {
            sbInt.insert(0, ' ');
        }
        return sbInt.toString();
    }

    public void printOneReservation () {
        calculateAmount ();
        sb.delete (0, sb.length());
        sb.append ("\nReservation:       ");
        sb.append (   intTo12String (reservationNumber) );
        sb.append ("\nNumber of seats:   ");
        sb.append (   intTo12String (seats) );
        sb.append ("\nNumber of days:    ");
        sb.append (   intTo12String (numberOfDays) );
        sb.append ("\nDay rate per seat: ");
        sb.append (   formatMoney(dayRatePerSeat));
        sb.append ("\nRoom amount:       ");
        sb.append (   formatMoney(roomAmount) + "\n");
        System.out.println (sb.toString());
    }

    public int getReservationNumber () {
        return reservationNumber;
    }
    public void setReservationNumber(int reservationNumber) {
        sb.delete(0, sb.length());
        String s = Integer.toString (reservationNumber);
/*1*/   if (s.length() != 6) {
            sb.append ("invalid length=");
            sb.append (s.length());
            sb.append ("\n");
        }
/*2*/   if (! s.startsWith ("130") ) {
            sb.append ("does not start with 130\n");
        }
/*3*/   char c3 = s.charAt (3);
        if (c3 == s.charAt(4) && c3 == s.charAt(5) ) {
            sb.append ("chars 4, 5, and 6 are the same\n");
        }
        if (sb.length() == 0) {
            this.reservationNumber = reservationNumber;
        } else {
```

```
                sb.insert (0, "\n");
                sb.insert (0, DEFAULT_RESERVATION_NUMBER);
                sb.insert (0, " is invalid, will use ");
                sb.insert (0, reservationNumber);
                sb.insert (0, "\n");
                System.err.println (sb.toString() );
                this.reservationNumber =
                    DEFAULT_RESERVATION_NUMBER;
            }
    }

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

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

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



```

// ===============================Start of new code=====
// ===============================toString
// ===============================hashCode
// ===============================equals
// ===============================End of new code=======
}

// Starter code for RoomResWithFood13.java --------------------
 package com.themisinc.u13;
 public class StarterRoomResWithFood13
   extends RoomReservation13 {
     public static final double AM_COST_PER_PERSON=9.00;
     public static final double PM_COST_PER_PERSON=8.00;

     private boolean amService;
     private boolean pmService;

     private double foodAmount;
     private StringBuilder sBldr = new StringBuilder();

     public RoomResWithFood13 () {
     }

     public RoomResWithFood13(
       int reservationNumber, int seats,
       int numberOfDays, double dayRatePerSeat,
       boolean amService, boolean pmService
     ) {
         super (reservationNumber, seats,
              numberOfDays, dayRatePerSeat);
         setAmService (amService);
         setPmService (pmService);
     }

     public RoomResWithFood13(
       int reservationNumber, int seats,
       int numberOfDays,
       boolean amService, boolean pmService
     ) {
         this (reservationNumber, seats,
         numberOfDays,
         RoomReservation13.DEFAULT_DAY_RATE_PER_SEAT,
         amService, pmService);
     }

     private void calculateAmount () {
         double perDay = 0.0;
         if (isAmService() ) {
             perDay = AM_COST_PER_PERSON;
         }
         if (isPmService() ) {
             perDay = perDay + PM_COST_PER_PERSON;
         }
```

```
        foodAmount = getSeats() * getNumberOfDays() * perDay;
    }

    public void printOneReservation () {
        calculateAmount();
        super.printOneReservation();
        sBldr.delete (0, sBldr.length() );
        sBldr.append ("**Food Charges:");
        sBldr.append ("\n  Food:            ");
        sBldr.append (formatMoney(foodAmount) );
        sBldr.append ("\n");
        System.out.println (sBldr.toString() );
    }

    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;
    }

// ===============================Start of new code=====
// ===============================toString
// ===============================hashCode
// ===============================equals
// ===============================End of new code=======
}
```


(blank)

UNIT 14: COLLECTIONS FRAMEWORK: LISTS

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

1. Briefly describe and compare characteristics of arrays and the classes Vector, ArrayList, and LinkedList.
2. Use the classes Vector, ArrayList, and LinkedList to store, retrieve, and manipulate objects of different classes.
3. Briefly describe and compare characteristics of the interfaces Collection and List.
4. Briefly describe and compare characteristics of Enumeration, Iterator, and ListIterator.
5. Use Enumerations, Iterators, and ListIterators to traverse the elements of Vector, ArrayList, and LinkedList.

Classes

Interfaces

CONTAINERS, COLLECTIONS

1. A container is an object that can contain zero or more variables. Usually the term is used when the variables are references pointing to other objects.

   a. Usually a container class has methods to add, remove, and access the contained objects.

   b. Some container classes have methods to sort or otherwise rearrange the contained objects.

2. The term collection usually refers to a container of a group of similar objects, and the references are called elements.

3. Java 1 containers include arrays and the classes Vector, Stack, and Hashtable. These classes are not deprecated, but are used primarily in legacy code.

4. Java 1.2, called Java 2, introduced the Collections Framework, now known as the legacy Collections Framework, a group of interfaces and implementing classes for representing and manipulating collections. All classes and interfaces in the Collections Framework are in the package java.util.

5. Java 1.5, called Java 5, modified the Collections Framework by introducing Generics and new classes and interfaces. Until we cover generics in Unit 16, suppress compiler warnings via @SuppressWarnings ("unchecked")

6. Comparison of collection classes and arrays:

   a. The number of elements in an array is unchangeable after the array is constructed. The number of elements in a collection does not have to be specified when it is constructed, and can grow and/or shrink afterward.

   b. Array elements can be basic type or references of any class type, but all elements must be the same type. If an array contains Object references, each element can point to an object of any class type. Collection elements are always Object references, and each element can point to an object of any class type.

7. Array advantages are fast processing and efficient use of RAM. When data can be easily arranged into a list or table, an array may be the simplest form of container.

8. The array disadvantage is lack of methods for common functions. Because arrays have fixed size, to change the array size you must write your own code to create another array and System.arraycopy elements from one to the other.

TERMINOLOGY, SYNCHRONIZATION, FAIL-FAST

1. Terminology:

    a. Collections Framework, the entire framework in java.util.
    b. collection, a group of similar objects
    c. Collection, an interface in the Collections Framework.
    d. Collections, a class with static methods that are useful with classes of the Collections Framework.

2. Synchronization for Thread-safe use:

    a. Java 1 Collections classes Vector, Stack, and Hashtable are synchronized for Thread-safe use, resulting in greater overhead and slower performance.

    b. Java 2 introduced Collections classes with the same functionality without the overhead of Thread-safety.

Java 1, Thread-safe	Java2, Not Thread-safe
Vector	ArrayList
Stack	LinkedList
Hashtable	HashMap

    c. In Java EE, thread safety is handled by the server, so synchonization is not a issue.

    d. When an unsynchronized Collection, List, Set, or Map is constructed it can be wrapped in a synchronized object so that multiple Threads can be prevented from concurrently adding or removing elements, or resizing.

    ```
    List a = Collections.synchronizedList(new ArrayList());
    ```

3. Fail-fast:

    a. Java 1 containers provide iteration by means of an Enumeration. If you modify the container by adding or deleting elements while enumerating, the Enumeration does not notice, but your results might be corrupted.

    b. Java 2 introduced Iterator which is fail-fast and also provides a remove method. Fail-fast means if the collection is modified after the iterator is created (except through the iterator's remove method) the iterator throws ConcurrentModificationException. The purpose of fail-fast is to prevent the risk of unpredictable results. If unsynchronized concurrent modification is done, fail-fast is not guaranteed.

Vector

1. The Vector class implements the List interface, and contains a structure similar to an array of Object references. Vector elements are accessed via an integer index starting at zero. Vector allows duplicate and null elements.

2. Vector capacity is the total number of element slots, and size is the number of element slots that contain objects.

3. Vector constructors allow you to specify initial capacity and growth increment. Default initial capacity is 10. Capacity doubles each time new space is needed
   a. Vector v = new Vector();
   b. Vector v = new Vector(initialCapacity);
   c. Vector v = new Vector(initialCapacity, growthIncrement);

4. When Java 2 introduced the Collections Framework, Vector was given new, shorter method names, and Iterator replaced Enumeration as a better way to loop through the elements.

Java 1	Java 2	Method function
addElement	add	append element to end of Vector
removeElement	remove	remove specified element
elementAt	get	return specified element
elements	iterator	return Enumeration or Iterator

5. Some public instance methods of Vector:

```
a.  void add(Object o)            //Add o to end of Vector
b.  void add(int index, Object o) //Add o at index location
c.  int capacity()                //Return capacity
d.  void clear()                  //Remove all elements
e.  boolean contains()            //True if element exists
f.  Object firstElement()         //return element[0]
g.  Object get(int index)         //return element[index]
h.  int indexOf(Object elem)      //return index of elem
i.  Object lastElement()          //return element[size-1]
f.  void remove(int index)        //remove element[index]
g.  int size();                   //return size
h.  void trimToSize();            //shrink capacity to size
j.  Iterator iterator();          //return an Iterator
k.  Enumeration elements();       //return an Enumeration
```

Result, AJ1405.java

```
1. capacity=4, size=3
2. capacity=3, size=3
3. 0=true, 1=one string, 2=[Ljava.lang.String;@1db9742
0=true  1=[Ljava.lang.String;@1db9742
true  [Ljava.lang.String;@1db9742
true  [Ljava.lang.String;@1db9742
true  [Ljava.lang.String;@1db9742  Hello Java
```

Vector EXAMPLE

AJ1405.java

```
import java.util.Vector;
import java.util.Enumeration;
import java.util.Iterator;
@SuppressWarnings ("unchecked")
public class AJ1405 {
    public static void main (String[] args) {

/*1*/   Vector v = new Vector (2, 2); //capacity,growthIncr
        v.add (new Boolean(true) );          //0
        v.add (new String("one string") );   //1
        String[] arrayObj = {"Hello", "Java"};
        v.add (arrayObj);                    //2

/*2*/   System.out.println ("1. capacity=" + v.capacity() +
            ", size=" + v.size());
        v.trimToSize();
        System.out.println ("2. capacity=" + v.capacity() +
            ", size=" + v.size());

/*3*/   System.out.println ("3. 0=" + v.firstElement() +
            ", 1=" + v.get(1) +
            ", 2=" + v.lastElement() );

/*4*/   v.remove(1);

/*5*/   for (int i=0; i<v.size(); i++)
            System.out.print (i+"=" + v.get(i) + "  ");
        System.out.println ();

/*6*/   for (Enumeration e=v.elements();e.hasMoreElements();)
            System.out.print(e.nextElement() + "  ");
        System.out.println ();

/*7*/   for (Iterator i = v.iterator(); i.hasNext(); )
            System.out.print(i.next() + "  ");
        System.out.println ();

/*8*/   for (Object elem : v) {
            System.out.print (elem + "  ");
/*9*/       if (elem instanceof String[]) {
                String[] sArray = (String[]) elem;
                for (String s : sArray) {
                    System.out.print (s + " ");
                }
            }
        }
        System.out.println ();
    }
}
```


Vector CatalogPage EXAMPLE

AJ1406.java

```
public class AJ1406 {
    public static void main (String[] args) {

        DyeColor dc;
        CatalogPage1406 cp = new CatalogPage1406 ();

        //SOCKS
        dc = new DyeColor("RED", 10.60);
        if (cp.add("socks", dc) == false) pErr ("RED");
        dc = new DyeColor("YELLOW", 8.20);
        if (cp.add("socks",dc) == false) pErr ("YELLOW");

        //TIES
        dc = new DyeColor("RED", 10.60);
        if (cp.add("ties", dc) == false) pErr ("RED");
        dc = new DyeColor("GREEN", 7.50);
        if (cp.add("ties", dc) == false) pErr ("GREEN");
        dc = new DyeColor("BLUE", 12.90);
        if (cp.add("ties", dc) == false) pErr ("BLUE");

        cp.printData();
    }
    private static void pErr (String s) {
        System.err.println (s + ", add failed, exiting");
        System.exit (1);
    }
}
```

CatalogPage1406.java

```
import java.util.Vector;
import java.util.Iterator;
@SuppressWarnings ( "unchecked" )
public class CatalogPage1406 {             //Vector Version

    private Vector socksVector = new Vector();
    private Vector tiesVector = new Vector();

    public CatalogPage1406() {
    }

    public boolean add (String garment, DyeColor dc){
        if (garment == null || dc == null) {
            return false;
        }
        boolean returnValue = false;
        if (garment.equals("socks") ) {
            socksVector.add (dc);
            returnValue = true;
        }
```

```
        if (garment.equals("ties") ) {
            tiesVector.add (dc);
            returnValue = true;
        }
        return returnValue;
    }

    public void printData () {

        //runtime polymorphism in both println statements

        for(Iterator i=socksVector.iterator(); i.hasNext();){
            System.out.println ("socks=" + i.next() );
        }
        for (Object o : tiesVector) {
            System.out.println ("ties=" + o);
        }
    }
}
```

DyeColor.java

```
public class DyeColor {
    private String color;
    private double price;

    public DyeColor (String color, double price) {
        setColor (color);
        setPrice (price);
    }

    public String getColor () {
        return color;
    }
    public void setColor (String color) {
        this.color = color;
    }
    public double getPrice () {
        return price;
    }
    public void setPrice (double price) {
        this.price = price;
    }

    public String toString() {
        return "DyeColor[" + color + ", " + price + "]";
    }
}
```

Result, AJ1406.java

```
socks=DyeColor[RED, 10.6]
socks=DyeColor[YELLOW, 8.2]
ties=DyeColor[RED, 10.6]
ties=DyeColor[GREEN, 7.5]
ties=DyeColor[BLUE, 12.9]
```


ArrayList

1. ArrayList is a resizable array that implements the List interface.

    a. ArrayList provides the functionality of Vector except ArrayList is not synchronized for Thread-safe use, and must use an iterator rather than an Enumeration.

    b. ArrayList permits duplicate and null elements.

2. ArrayList capacity grows automatically if needed. Some methods enable you to change capacity.

    a. If constructed without a specified initial capacity, the default initial capacity is 10. <u>Capacity</u> is the number of slots, which is at least as large as <u>size</u> which is the number of slots containing elements.

    b. ensureCapacity(int minCapacity); Increases capacity if needed to hold at least the minCapacity. If you increase capacity before adding a large number of elements, this can reduce the amount of reallocation.

    c. trimToSize(); Trims capacity to the list's current size.

3. Some ArrayList methods:

    a. add(element); Append element to end of this list.
    b. addAll(collection); Append all elements in collection to this list.
    c. clear(); Remove all elements from this list.
    d. contains(object); Return true if list contains object.
    e. get(index); Return element at index.
    f. indexOf(object); Return index of first occurrence of object, or -1 if not contained in list.
    g. iterator(); Return an Iterator for this ArrayList.
    h. lastIndexOf(object); Return index of last occurrence of object, or -1 if not contained in list.
    i. isEmpty(); Return true if list contains no elements.
    j. remove(index); Remove element at index.
    k. remove(object); Remove first occurrence of object if contained in list.
    l. set(index, elem); Replace element at index with elem.
    m. size(); Return number of elements.
    n. toArray(); Return Object array with all elements of list.

ArrayList EXAMPLE

AJ1409.java
```
import java.util.ArrayList;
import java.util.Iterator;
@SuppressWarnings ("unchecked")

public class AJ1409 {
    public static void main (String [] args) {

/*1*/   ArrayList a = new ArrayList ();
        a.add ("1");
        a.add (null);                      //allows null elements
        a.add (2);                                   //autoboxing
        a.add (new Integer(2));           //allows duplicates

        System.out.println ("size=" + a.size() + ": " + a);

/*2*/   for (int i=0; i<a.size(); i++) {
            System.out.print (a.get(i) + "  ");
        }
        System.out.println ();

/*3*/   for (Iterator i = a.iterator(); i.hasNext(); ) {
            System.out.print(i.next() + "  ");
        }
        System.out.println ();
    }
}

```

Result, AJ1409.java
```
size=4: [1, null, 2, 2]
1  null  2  2
1  null  2  2
```

==

1. A reference to an ArrayList can be:

```
a.  ArrayList aList = new ArrayList ();  //class type ref
b.  List list = new ArrayList ();        //interface type
c.  Collection coll = new ArrayList ();  //interface type
```


LinkedList

1. Linkedlist implements the List interface. LinkedList permits duplicate and null elements.

    a. LinkedList is implemented as a doubly-linked list. Operations with indexes traverse the list from beginning or end, whichever is closer to the specified index.

    b. LinkedList has methods to get, remove, and insert an element at the beginning or end of the list. This enables a LinkedList to be used for a stack aka LIFO, queue aka FIFO, or deque (double-ended queue).

2. The constructors of LinkedList do not allow specifying the initial capacity.

3. Methods specific to LinkedList include:

```
a.  void addFirst(Object o); Add o as initial element.
b.  void addLast(Object o);  Add o as last element.
c.  Object getFirst();       Return initial element as an Object.
d.  Object getLast();        Return last element as an Object.
e.  Object removeFirst();    Remove and return head element.
f.  Object removeLast();     Remove and return last element.
g.  Type[] toArray(Type[]);  Return Type array with all elements.
```

4. LinkedList methods added in Java 1.5:

    a. Object peek(); Return element 0 and do not remove it.
    b. Object poll(); Return element 0 and remove it.

5. LinkedList methods added in Java 1.6:

    a. Object peekFirst(); Same as peek().
    b. Object peekLast(); Return and do not remove the last element, or return null if list is empty.

OPTIONAL

6. LinkedList is not synchronized.
7. Iterators for LinkedList are fail-fast.

LinkedList LIFO STACK EXAMPLE

AJ1411.java

```
public class AJ1411 {
    public static void main (String[] args) {
        LIFO stack = new LIFO();

        stack.pushLifo ("zero");
        stack.pushLifo ("one");
        stack.pushLifo (new Integer(2) );

        while ( ! stack.isEmpty() )
            System.out.println ("pop=" + stack.popLifo());
    }
}
```

LIFO.java

```
import java.util.LinkedList;
@SuppressWarnings ("unchecked")
public class LIFO {
    private LinkedList lifo = new LinkedList ();

    public void pushLifo (Object o) {
        lifo.addLast (o);
    }
    public Object popLifo () {
        Object o = null;
        if (lifo.size() != 0) {
            o = lifo.getLast();
            lifo.removeLast();
        }
        return o;
    }
    public boolean isEmpty () {
        if (lifo.size() == 0)
            return true;
        else
            return false;
    }
}
```

Result, AJ1411.java

```
pop=2
pop=one
pop=zero
```


OPTIONAL: Stack LIFO STACK EXAMPLE

AJ1412.java

```
import java.util.Stack;
import java.util.Iterator;
@SuppressWarnings ("unchecked")

public class AJ1412 {
    public static void main (String[] args) {

        Integer intObj    = new Integer (12);
        Character charObj = new Character ('a');
        String strObj     = new String ("Java");

        Stack s = new Stack ();
        System.out.println ("1. capacity=" + s.capacity() +
            ", size=" + s.size() + ", empty=" + s.empty());

        s.push (intObj);
        s.push (charObj);
        s.push (strObj);
        System.out.println ("2. capacity=" + s.capacity() +
            ", size=" + s.size() + ", empty=" + s.empty());

        System.out.println ("3. search=" + s.search("Java") +
            ", pop=" + s.pop() + ", peek=" + s.peek() );

        for (Iterator i = s.iterator();  i.hasNext();  )
            System.out.println ("iterator=" + i.next() );
    }
}
```

Result, AJ1412.java

```
1. capacity=10, size=0, empty=true
2. capacity=10, size=3, empty=false
3. search=1, pop=Java, peek=a
iterator=12
iterator=a
```

==

1. Stack in java.util extends Vector, and adds five methods so you can easily create and manipulate a LIFO stack. However, the class LinkedList is usually used instead of Stack.

    a. push adds an object to the top of the stack.
    b. pop removes the object on top of the stack.
    c. peek returns the top object but does not pop it.
    d. search returns an object's offset from the top of the stack or -1 if not found. The top offset is 1.
    e. empty returns true if the stack contains no objects.

OPTIONAL: QUEUES

1. A queue is a collection designed to store elements while they wait to be processed, typically in FIFO sequence (first-in, first-out). In a FIFO queue, new elements are inserted at the tail of the queue.

   a. A priority queue sequences its elements according to a specified Comparator object and its compare method, or the queue's own compareTo method if its objects implement Comparable.

   b. A LIFO queue aka stack sequences its elements last-in, first-out.

   c. Regardless of the sequencing used, the "head" of a queue is the element that will be removed by a call to remove() or poll(). Different types of queues use different placement rules, but every Queue implementation must specify its ordering properties.

2. Queues provide insertion, extraction, and inspection methods not required by the Collection interface. These methods either throw an exception or return null or false if the operation fails.

	Throw exception	Return null or false
Insert	add(e)	offer(e)
Remove	remove()	poll()
Examine	element()	peek()

   a. The method offer() inserts an element if possible or returns false, unlike the Collection.add method that can throws an unchecked exception if it cannot add the element. The offer method is used when failure is normal rather than exceptional, such as in fixed-capacity (or "bounded") queues.

   b. The remove() and poll() methods remove and return the head of the queue. Which element is the head depends on the queue's ordering policy. When the queue is empty remove()throws an exception, but poll() returns null.

   c. The element() and peek() methods return, but do not remove, the head of the queue.

3. Even if an implementation of Queue allows null elements, null elements should not be used because null is returned by the poll method to indicate that the queue contains no elements.

COLLECTIONS FRAMEWORK: CLASSES AND INTERFACES

1. The Collections Framework's interfaces and classes standardize the methods that handle collections. This allows the objects in a collection to be handled via the same methods regardless of the type of collection they are in.

2. Elements of all collections are references of type Object.

   a. To include an element that is a variable of a basic type, the variable must be stored in a wrapper class object.

   b. Autoboxing, introduced in Java 5, enables code to be written as if basic types were allowed as elements.

   c. After you retrieve a reference from a collection, to access its variables and methods (other than the ones defined in, and inherited from, Object), use instanceof to determine the actual class of the object, and then cast the reference to that type. This can be error-prone, and is addressed by generics and Java 5 Collections.

3. By convention, new collection classes should have at least two constructors:

   a. a no-argument constructor that creates a collection with no elements

   b. a copy-constructor with one argument of type Collection that copies the elements of any Collection into a new Collection of the constructor's class type.

4. Some interfaces and implementing classes (two views):

   a. *Collection (no sequence, dups ok)* (interfaces are *italic underlined*)
      1. *List (sequence, dups ok)*
         a) Vector
         b) ArrayList
         c) LinkedList
      2. *Set (no sequence, no dups)*
         a) HashSet
   b. *Map (key-value pairs)*
      1. Hashtable (random order via hashing algorithm)
      2. HashMap (random order via hashing algorithm)
   c. *Iterator (forward only)*
      1. *ListIterator (forward or backward)*

	Interface	Sequence	Duplicates
d.	Collection	no	allowed
e.	List	yes	allowed
f.	Set	no	no
g.	SortedSet	yes	no

Collection INTERFACE

1. Collection is the top interface in the Collections Framework.

2. No class in the Collections Framework implements Collection directly, but many classes implement its subinterfaces.

   a. A reference of type Collection is useful because it can point to objects with any type of Collection subinterface or their implementing classes.

   b. If you create a new class for an unordered collection that is allowed to have duplicate elements, called a "bag" or "multiset", it should implement Collection directly.

3. Some Collection methods:

   a. boolean add(Object o); Guarantee that o is in the collection, and return true if the collection was modified. If a subinterface of Collection does not allow duplicates, and already has the element, the element is not added and the method returns false. If a collection does not add the element because it violates any other rule of the collection, the method must throw an Exception.

   b. boolean contains(Object o); Return true if this collection contains the specified element.

   c. boolean containsAll(Collection c); Return true if this collection contains all of the elements in c.

   d. boolean isEmpty(); Return true if this collection contains no elements.

   e. Iterator iterator(); Return an iterator for the elements in this collection.

   f. boolean remove(Object o); Remove one instance of o if it is in this collection, and return true if the collection was modified.

   g. int size(); Return the number of elements in collection.

List INTERFACE

1. A List is an ordered collection, also called a sequence. The List interface extends (is a subinterface of) Collection.

    a. Lists allow you to control where an element is inserted, access elements by index position, and search the List for a specific element.

    b. Element indexes start with zero.

    c. List methods allow inserting and removing multiple elements at a specified index location in the list.

    d. Lists usually allow duplicates, as well as multiple null elements if null elements are allowed.

    e. The methods iterator, add, remove, equals, and hashCode are defined with more requirements in List than in Collection.

2. Lists can but should not contain Lists as elements, because then the equals and hashCode methods may not work properly.

3. Iterating over list elements may be faster than looping through the list via the use of index numbers.

4. Some List methods:

    a. boolean add(Object o); Append o to the end of List, and return true if List is modified.

    b. void add(int index, Object o); Insert o at index position and move current element at index and subsequent elements to the right.

    c. Object remove(int index); Remove element at index position and return element.

    d. boolean remove(Object o); Remove first instance of o (the one with the lowest index) if it is in List, and return true if List is changed.

    e. Object get(int index); Return element at index position.

    f. int indexOf(Object o); Return index of first occurrence of o, or -1 if o is not in List.

Iterator INTERFACE

1. Java 2 introduced Iterator to replace Enumeration. Iterator and Enumeration differ in these ways:

    a. Iterator method names are shorter.

    b. Modifying a collection while traversing it:

        1) You should not modify a collection while looping through it with an Enumeration.

        2) Iterators allow you to remove the most recently returned element during iteration via the Iterator remove() method.

2. The Iterator interface defines three methods for retrieving one element at a time from a Collection.

    a. boolean hasNext(); Return true if the Collection still has one or more elements to be returned.

    b. Object next(); Return the next element.

    c. void remove(); Remove the element that was returned from the immediately previous call to the next() method. This method may be called only once per call to next(). Unspecified behavior occurs if the collection is modified during iteration other than via remove().

3. An Iterator object is associated with a specific Collection object. The Iterator is obtained by calling the Collection object's iterator method.

4. The sequence of elements returned by an Iterator is:

    a. For a Set, which has no sequence: unspecified order.
    b. For a List, which has sequence: in forward sequence.

5. Iterator has one subinterface, ListIterator, which can return elements of a List in forward or backward sequential order.

ListIterator INTERFACE

1. ListIterator is a subinterface of Iterator. ListIterator differs from Iterator in these ways:

    a. can add, replace, or delete elements in the List
    b. has bidirectional access forward or backward
    c. can obtain the iterator's "cursor position" in the List

2. In a list of three elements 0, 1 and 2, a ListIterator's cursor position can be:

    a. 0 (before the element 0)
    b. 1 (between elements 0 and 1)
    c. 2 (between elements 1 and 2)
    d. 3 (after element 2)

3. ListIterator has nine methods:

    a. void add(Object o); Insert o before the element that would be returned by the next call to next(), and after the element that would be returned by the next call to previous().

    b. boolean hasNext(); Return true if there are more elements in the forward direction.

    c. boolean hasPrevious(); Return true if there are more elements in the backward direction.

    d. Object next(); Return the next element.

    e. int nextIndex(); Return the index of the element that would be returned by the next call to next().

    f. Object previous(); Return the previous element. Alternate calls to next() and previous() will return the same element repeatedly. Calling previous() after add() will return the element just added.

    g. int previousIndex(); Return the index of the element that would be returned by the next call to previous().

    h. void remove(); Remove the element that was returned by the most recent call to next() or previous().

    i. void set(Object o); Replace the element that was returned by the most recent call to next() or previous() with o, if allowed by the rules of the list.

ListIterator INTERFACE, EXAMPLE

AJ1419.java

```
import java.util.List;
import java.util.ArrayList;
import java.util.ListIterator;
@SuppressWarnings ("unchecked")

public class AJ1419 {
    public static void main (String [] args) {

        List a = new ArrayList ();
        a.add ("00");
        a.add ("11");
        a.add ("22");
        p ("1. ArrayList elements:  " + a + "\n");

        ListIterator it = a.listIterator();   //if a is type
                                              //Collection it
        p ("2. Forward:  ");                  //can't get a
        while (it.hasNext() ) {               //ListIterator
            p (it.nextIndex() + "=" + it.next() + "  ");
        }

        p ("\n3. Forward again:  ");
        while (it.hasNext() ) {
            p (it.nextIndex() + "=" + it.next() + "  ");
        }

        p ("\n4. Backward:  ");
        while (it.hasPrevious() ) {
            p (it.previousIndex()+"="+it.previous()+"  ");
        }

        p ("\n5. Forward with for-each:  ");
        for (Object obj : a) {
            p ("  " + (String)obj);
        }
        p ("\n");
    }
    public static void p (String s) {
        System.out.print (s);
    }
}
```

Result, AJ1419.java

```
1. ArrayList elements:  [00, 11, 22]
2. Forward:  0=00  1=11  2=22
3. Forward again:
4. Backward:  2=22  1=11  0=00
5. Forward with foreach:    00  11  22
```


EXERCISES

1. StarterCode141.java below is in package com.themisinc.u14. Copy it to create CaseStudy141.java

   Copy the code and modify it to make two separate Vectors that contain reservation numbers only, one Vector for reservations with 10 seats, and the other for reservations with 14 seats.

   Then print the reservation numbers in each Vector. Use the class RoomReservation5.java without changes.

```
package com.themisinc.u14;
public class StarterCode141 {
    public static void main (String[] args) {

        RoomReservation5[] rrArray = {
            new RoomReservation5 (130321, 14, 5, 25.00),
            new RoomReservation5 (130322, 10, 2, 25.00),

            new RoomReservation5 (130323, 12, 5, 25.00),
            new RoomReservation5 (130324, 14, 1, 35.00),

            new RoomReservation5 (130325, 10, 4, 25.00),
            new RoomReservation5 (130326, 12, 2, 25.00),

            new RoomReservation5 (130327, 14, 5, 45.00),
            new RoomReservation5 (130328, 14, 3, 35.00),
        };
    }
}
```

   <u>Result, CaseStudy141.java in com.themisinc.u14</u>

```
Reservations with 10 seats
1. 130322
2. 130325
Reservations with 14 seats
1. 130321
2. 130324
3. 130327
4. 130328
```

2. Copy CaseStudy141.java twice and call the copies CaseStudy142.java and CaseStudy143.java. Use ArrayList instead of Vector in CaseStudy142, and LinkedList instead of Vector in CaseStudy143, and produce similar results.

3. OPTIONAL. Create a LIFO stack program using Vector, similar to the programs on page aj14.11 and aj14.12.

SOLUTIONS

<u>CaseStudy141.java in com.themisinc.u14</u>

```
package com.themisinc.u14;
import java.util.Collection;
import java.util.Vector;
import java.util.Iterator;
@SuppressWarnings ("unchecked")
public class CaseStudy141 {

    public static void main (String[] args) {

        RoomReservation5[] rrArray = {
            new RoomReservation5 (130321, 14, 5, 25.00),
            new RoomReservation5 (130322, 10, 2, 25.00),

            new RoomReservation5 (130323, 12, 5, 25.00),
            new RoomReservation5 (130324, 14, 1, 35.00),

            new RoomReservation5 (130325, 10, 4, 25.00),
            new RoomReservation5 (130326, 12, 2, 25.00),

            new RoomReservation5 (130327, 14, 5, 45.00),
            new RoomReservation5 (130328, 14, 3, 35.00),
       };

        Collection c10 = new Vector ();
        Collection c14 = new Vector ();

        for (RoomReservation5 rr : rrArray) {
            if (rr.getSeats() == 10) {
                c10.add (rr.getReservationNumber() );
            }
            if (rr.getSeats() == 14) {
                c14.add (rr.getReservationNumber() );
            }
        }

        int lineNo = 1;
        System.out.println ("Reservations with 10 seats");
        for (Iterator i = c10.iterator(); i.hasNext(); ) {
            System.out.println (lineNo++ + ". " + i.next());
        }

        lineNo = 1;
        System.out.println ("Reservations with 14 seats");
        for (Object o : c14) {
            System.out.println (lineNo++ + ". " + o);
        }
    }
}
```


CaseStudy142.java in com.themisinc.u14

```java
package com.themisinc.u14;
import java.util.Collection;
import java.util.ArrayList;
import java.util.Iterator;
@SuppressWarnings ("unchecked")
public class CaseStudy142 {

    public static void main (String[] args) {

        RoomReservation5[] rrArray = {
            new RoomReservation5 (130321, 14, 5, 25.00),
            new RoomReservation5 (130322, 10, 2, 25.00),

            new RoomReservation5 (130323, 12, 5, 25.00),
            new RoomReservation5 (130324, 14, 1, 35.00),

            new RoomReservation5 (130325, 10, 4, 25.00),
            new RoomReservation5 (130326, 12, 2, 25.00),

            new RoomReservation5 (130327, 14, 5, 45.00),
            new RoomReservation5 (130328, 14, 3, 35.00),
       };

        Collection c10 = new ArrayList ();
        Collection c14 = new ArrayList ();

        for (RoomReservation5 rr : rrArray) {
            if (rr.getSeats() == 10) {
                c10.add (rr.getReservationNumber() );
            }
            if (rr.getSeats() == 14) {
                c14.add (rr.getReservationNumber() );
            }
        }

        int lineNo = 1;
        System.out.println ("Reservations with 10 seats");
        for (Iterator i = c10.iterator(); i.hasNext(); ) {
            System.out.println (lineNo++ + ". " + i.next());
        }

        lineNo = 1;
        System.out.println ("Reservations with 14 seats");
        for (Object o : c14) {
            System.out.println (lineNo++ + ". " + o);
        }
    }
}
```


CaseStudy143.java in com.themisinc.u14

```java
package com.themisinc.u14;
import java.util.Collection;
import java.util.LinkedList;
import java.util.Iterator;
@SuppressWarnings ("unchecked")
public class CaseStudy143 {

    public static void main (String[] args) {

        RoomReservation5[] rrArray = {
            new RoomReservation5 (130321, 14, 5, 25.00),
            new RoomReservation5 (130322, 10, 2, 25.00),

            new RoomReservation5 (130323, 12, 5, 25.00),
            new RoomReservation5 (130324, 14, 1, 35.00),

            new RoomReservation5 (130325, 10, 4, 25.00),
            new RoomReservation5 (130326, 12, 2, 25.00),

            new RoomReservation5 (130327, 14, 5, 45.00),
            new RoomReservation5 (130328, 14, 3, 35.00),
       };

        Collection c10 = new LinkedList ();
        Collection c14 = new LinkedList ();

        for (RoomReservation5 rr : rrArray) {
            if (rr.getSeats() == 10) {
                c10.add (rr.getReservationNumber() );
            }
            if (rr.getSeats() == 14) {
                c14.add (rr.getReservationNumber() );
            }
        }

        int lineNo = 1;
        System.out.println ("Reservations with 10 seats");
        for (Iterator i = c10.iterator(); i.hasNext(); ) {
            System.out.println (lineNo++ + ". " + i.next());
        }

        lineNo = 1;
        System.out.println ("Reservations with 14 seats");
        for (Object o : c14) {
            System.out.println (lineNo++ + ". " + o);
        }
    }
}
```


FIFO QUEUE USING Vector

<u>VectorFIFO.java</u>

```
package com.themisinc.u14;
import java.util.Vector;
@SuppressWarnings ("unchecked")
public class VectorFIFO {
    public static void main (String[] args) {
        FirstInFirstOut fifo = new FirstInFirstOut (4);
        System.out.println ("1. retrieve: " + fifo.get() );
        fifo.add ("zero");
        fifo.add ("one");
        fifo.add ("two");
        while (! fifo.isEmptyQueue() )  {
            String s = (String) fifo.get();  //returns Object
            System.out.println ("2. retrieve: " + s);
        }
    }
}
class FirstInFirstOut {
    private Vector v;
    private int numberOfElements;
    public FirstInFirstOut (int initialCapacity) {
        v = new Vector (initialCapacity);
    }
    public Object add (Object newElement) {
        v.add (newElement);
        numberOfElements++;
        return newElement;
    }
    public Object get() {                //need synchronization
        Object o = null;                 //for multiple users
        if (v.size() != 0) {
            o = v.firstElement();
            v.remove(v.firstElement() );
            numberOfElements--;
        }
        return o;
    }
    public boolean isEmptyQueue () {
        if (numberOfElements == 0) return true;
        else return false;
    }
}
```

<u>Result, VectorFIFO.java</u>

```
1. retrieve: null
2. retrieve: zero
2. retrieve: one
2. retrieve: two
```

UNIT 15: COLLECTIONS FRAMEWORK: MAPS

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

1. Briefly describe the difference between the interfaces List, Map, and Set.

2. Use the classes Hashtable, HashMap, and HashSet.

3. Use an Iterator to iterate over a Map or a Set.

Hashtable

1. A Hashtable contains key-value pairs in non-predictable order. The key and value are Object type references. Any unique, non-null object can be used as a key or value.

2. Initial capacity and load factor affect the performance of a Hashtable.
   a. Capacity is the number of buckets.
   b. Load factor is how full the Hashtable can get before capacity is increased automatically, but when and whether a Hashtable is rehashed are implementation-dependent.

3. If a hash collision occurs, a single bucket stores multiple entries, and the JVM searches them sequentially.

4. The default load factor is .75. If many entries will be made, creating the Hashtable with a large capacity may be more efficient than relying on automatic rehashing as it grows.

5. a. Finding a given element in a collection via sequential search requires iterating over an average of half the elements. The more elements, the longer the search takes.

   b. If each element is stored at a specific location via a hashing algorithm, each element can be retrieved via the same algorithm, achieving close to constant time performance regardless of number of elements in the collection.

6. Hashtable iterators try to be fail-fast but Enumerations returned by the keys() and elements() methods are not fail-fast (aj14.03). Hashtable, introduced in Java 1, is Thread-safe. HashMap, introduced in Java 2, is not (aj14.03).

7. Methods of Hashtable include:

a. clear() Clears the Hashtable so it contains no keys
b. containsKey(Object key) Returns true if the specified object is a key in this Hashtable
c. containsValue(Object value) Returns true if one or more keys map to value
d. elements() Returns an Enumeration of the values
e. get(Object key) Returns the value that key maps to, or null if key is not present
f. isEmpty() Returns true if there are no key-value pairs
g. keys() Returns an Enumeration of the keys
h. put(K key, V value) Puts the key-value pair in the Hashtable
i. remove(Object key) Removes the key and its value, and returns the value
j. size() Returns the int number of keys
k. values() Returns a Collection of the values, useful to get an Iterator for the values in the Hashtable.

Hashtable, EXAMPLE 1

AJ1503.java

```java
import java.util.Hashtable;
import java.util.Enumeration;
import java.util.Collection;
import java.util.Iterator;
@SuppressWarnings ("unchecked")

public class AJ1503 {
    public static void main (String[] args) {
        Integer i1 = new Integer(1);
        Integer i2 = new Integer(2);
        Integer i3 = new Integer(3);

        Hashtable h = new Hashtable ();
        if (h.isEmpty()) p ("1. empty");

        h.put ("one", i1 );             //keys are String.
        h.put ("two", i2 );             //values are Integer.
        h.put ("three", i3 );           //keys and values can
        p ("2. size=" + h.size());      //not be null.

        if (h.containsKey ("one") && h.containsValue(i2) )
            p ("3. 1-2");

        Object oValue = h.get("one");//Runtime Polymorphism
            p ("4. oValue=" + oValue);

        Integer iValue = (Integer) h.remove ("three");
            p ("5. iValue=" + iValue);

        Enumeration keys = h.keys();
        Enumeration values = h.elements();
        while (keys.hasMoreElements() )
            p ("6. " + keys.nextElement() + "=" +
             values.nextElement() );

        Collection c = h.values();
        for (Iterator i = c.iterator(); i.hasNext(); )
            p ("7. " + i.next() );

        h.clear();
        p ("8. size=" + h.size() + "\n");
    }
    public static void p (String s) {
        System.out.print (s + "    ");
    }
}
```

Result, AJ1503.java

```
1. empty    2. size=3    3. 1-2    4. oValue=1    5. iValue=3  6.
 two=2    6. one=1    7. 2    7. 1    8. size=0
```


Hashtable, EXAMPLE 2

AJ1504.java

```
import java.util.Hashtable;
@SuppressWarnings ("unchecked")
public class AJ1504 {
    public static void main (String [] args) {

        String [] numbers = {
            "914-456-1234",
            "212-123-1234",
            "415-778-1234",
            "914-654-1234",
            "914-724-1234",
            "415-224-1234"
        };

        Hashtable h = new Hashtable ();

        Integer refOldHowMany;
        int     intNewHowMany;

      //for (int i=0; i<numbers.length; i++) {
      //    String areaCode = numbers[i].substring(0,3);

        for (String num : numbers) {
            String areaCode = num.substring(0,3);

            if ( h.containsKey(areaCode) ) {

                refOldHowMany = (Integer) h.get(areaCode);

              //intNewHowMany=1 + refOldHowMany; //unboxing
                intNewHowMany=1 + refOldHowMany.intValue();

            } else {

                intNewHowMany = 1;
            }

          //h.put(areaCode, intNewHowMany);      //autoboxing
            h.put(areaCode, new Integer(intNewHowMany) );
        }

        System.out.println (h);
    }
}
```

Result, AJ1504.java

```
{415=2, 212=1, 914=3}
```


Map INTERFACE, HashMap, KEY-VALUE PAIRS

1. The Map interface is part of the Collections Framework, but it is not a subinterface of Collection. Map uses different method names, such as "put" instead of "add".

2. A map, also known as an associative array or hash table, is a collection of key-value pairs.

   a. Each map element has a key and its associated value.

   b. The keys and values are stored as Object references.

   c. <u>The keys must be unique (no duplicates). When you put an element with a key that already exists, the new value replaces the existing element's value.</u>

   d. Objects used as map keys should not be modified while being used as keys.

   e. Map gives you three ways to retrieve data from a map:
      1) "keys only" using keySet()
      2) "values only" using values()
      3) "key-value pairs" using entrySet(). You can also retrieve key-value pairs by retrieving the keys and then using each key to retrieve its value.

3. Classes that implement the Map interface include HashMap and HashSet. HashSet creates a HashMap of keys only (each key has a value that is a reference to the same dummy Object).

4. The class Hashtable originated in Java 1 and was revised in Java 2 to implement the Map interface. Hashtable is synchronized; HashMap is not.

OPTIONAL

5. The Map interface contains the inner static interface Map.Entry (a Map.Entry is a key-value pair).

6. HashMap may use a private inner class, HashIterator, that implements the Iterator interface.

7. Map does not specify whether the map must have sequence. TreeMap is ordered, but HashMap is unordered.

8. A new class that implements Map should have a constructor with no arguments that creates a map with no elements, and a copy constructor that receives one parameter of type Map and creates a new map with the same elements, as well as any other constructors. A new class that implements Map may restrict the keys and values. For example, it may disallow null values.

METHODS IN THE Map INTERFACE

1. Some Map methods:

   a. void clear(); Remove all mappings from this map.

   b. boolean containsKey(Object key); Return true if this map contains key.

   c. boolean containsValue(Object value); Return true if this Map maps one or more keys to value.

   d. Set entrySet(); Return a Set containing the elements in this map. Each element is type Map.Entry and has a key and its associated value. Elements can be removed but not added to the returned Set.

   e. Object get(Object key); Return the value for key or null if key is not in this Map (or maps to a null value). Use containsKey() to determine if key is in this map.

   f. boolean isEmpty(); Return true if this Map has no elements.

   g. Set keySet(); Return a Set containing the keys. Elements can be removed but not added to the returned Set.

   h. Object put(Object key, Object value); Put a new key-value element into this map, and return the previous value associated with key (or null if the key was not in this map or if the previous value was null).

   i. Object remove(Object key); Remove key and its value, and return its value (or null if the key was not in this map or had a null value).

   j. int size(); Return the number of key-value elements.

   k. Collection values(); Return a Collection of the values in this Map. Elements can be removed but not added to the returned Collection.

2. HashMap constructors allow you to specify initial capacity and load factor. The defaults are capacity 16 and load factor 0.75.

   a. HashMap h = new HashMap();
   b. HashMap h = new HashMap(int capacity);
   c. HashMap h = new HashMap(int capacity, float loadFactor);

HashMap, EXAMPLE

AJ1507.java

```
import java.util.HashMap;
import java.util.Map;
@SuppressWarnings ("unchecked")
public class AJ1507 {
    public static void main (String [] args) {

       String [] phoneNumbers = {
           "914-456-1234",
           "212-123-1234",
           "415-778-1234",
           "914-654-1234",
           "914-724-1234",
           "415-224-1234"
        };

        Map m = new HashMap ();

        Integer refOldHowMany;
        int     intNewHowMany;

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

            String areaCode = phoneNumbers[i].substring(0,3);

            if ( m.containsKey(areaCode) ) {

                refOldHowMany = (Integer) m.get(areaCode);

                intNewHowMany = 1 + refOldHowMany.intValue();

            } else {

                intNewHowMany = 1;
            }

            m.put(areaCode, new Integer(intNewHowMany) );
        }

        System.out.println (m);
    }
}
```

Result, AJ1507.java

```
{212=1, 914=3, 415=2}
```


Set INTERFACE AND HashSet CLASS

1. The Set interface is implemented by collections that have no duplicates and no sequence.
    a. Set constructors must not allow duplicates.
    b. One null element is allowed.
    c. A Set may not contain another Set.

2. Some Set methods:

    a. boolean add(Object o); Add o if it is not already in the Set, and return true if o is added.
    b. boolean equals(Object o); Compare this Set with o and return true if o is a Set with the same size, and every member of one Set is in the other.
    c. int hashCode(); Return an int that is the sum of the hash codes of all elements in the Set.

3. HashSet implements Set, and is a HashMap with keys only. Each key has a value that is a reference to the same dummy Object.

    a. HashSet differs from classes that implement List in that HashSet elements are unordered, have no first or last element, and have no index. There is no specific order when the HashSet is interated.

    b. The reference to a HashSet can be of type HashSet, Set, or Collection.

4. HashSets have constant time performance for the basic operations of add, remove, contains, and size, but speed of iterating depends on capacity (number of buckets) and load factor (number of elements). Do not set the initial capacity too high or the load factor too low for best iteration speed. Load factor is the percent of full buckets that triggers automatic capacity increase.

5. Some HashSet methods:

    a. boolean add(element); Add element if not already present and return true if added.
    b. void clear(); Remove all elements.
    c. boolean contains(object); Return true if object is in the HashSet.
    d. boolean isEmpty(); Return true if HashSet has no elements.
    e. Iterator iterator(); Return an iterator for this HashSet.
    f. boolean remove(element); Remove element if present and return true if removed.
    g. int size(); Return the number of elements.

6. Iterators for HashSet are fail-fast. HashSet is not synchronized for Thread-safety.

HashSet, EXAMPLE

AJ1509.java

```
import java.util.Collection;
import java.util.HashSet;
import java.util.Iterator;
@SuppressWarnings ("unchecked")

public class AJ1509 {
    public static void main (String [] args) {

        Collection c = new HashSet ();

        c.add ("1");
        c.add (new Integer(2));
        c.add (3);                           //autoboxing

        if (  c.add(new Double(4.5))  ) {    //returns true
            System.out.println ("1. added 4.5");
        }

        if (  c.add(4.5)  ) {    //autoboxing //returns false
            System.out.println ("2. added 4 again");
        }

        System.out.println ("3. size: " + c.size());
        System.out.println ("4. elements: " + c);

        for (Iterator i = c.iterator(); i.hasNext(); ) {
            System.out.print(i.next() + "  ");
        }
        System.out.println ();
    }
}
```

Result, AJ1509.java

```
1. added 4.5
3. size: 4
4. elements: [3, 2, 1, 4.5]
3  2  1  4.5
```

Result, AJ1509.java (different version of JDK)

```
1. added 4.5
3. size: 4
4. elements: [2, 1, 3, 4.5]
2  1  3  4.5
```


OPTIONAL: HASHING

1. A hash code, or hash, is a number derived by performing a hashing algorithm on data.

2. Two common uses of hashes are:

    a. Create algorithms to randomly store and retrieve data.

    b. "Summarize" the data in an object.

3. If a hashing algorithm is good, two objects containing the same data will get the same hash, and if two objects have different hashes it means their data is different.

4. A hash table is a collection for which a storage space is reserved, and divided into pieces called buckets or bins. A bucket holds one element. A hash method, using a hashing algorithm, assigns each element to a bucket, or uses the same algorithm later to locate the bucket to retrieve the element.

5. A hash table is a collection of buckets and a group of hashing methods that use a specific algorithm to put or get elements into or from their buckets.

6. Iteration order is unpredictable for all collections that are hashes. The Iteration order can change if elements are added or dropped, or if compiled or executed by a different version of Java.

OPTIONAL: Collections CLASS

1. The java.util.Collections class contains static methods to work with various collections classes, to manipulate a collection in various ways. These methods are overloaded.

   a. int binarySearch (List source, Object keyToSearchFor);
      Binary search algorithm for a specific element, which is useful with Objects that are keys in a TreeMap or elements in a TreeSet.

   b. void copy(List dest, List source);
      Copy elements from one list into another.

   c. void fill(List source, Object obj);
      "Fill" by replacing all elements in a list with one specified element.

   d. Object max (Collection coll);
      Return the "maximum" element of a collection, according to the natural ordering of its elements.

   e. Object min (Collection coll);
      Return the "minimum" element of a collection, according to the natural ordering of its elements.

   f. void reverse(List source);
      Reverse the sequence of elements.

   g. void shuffle(List source);
      Shuffle the elements in a list into random sequence.

   h. Sort, covered in Unit 19.

   i. void swap(List source, int i, int j);
      Swap the two elements at specified positions.

   j. List synchronizedList (List source);
      Wrap the collection in a thread-safe wrapper.

   k. Collection unmodifiableCollection (Collection c);
      Wrap the collection in an "unmodifiable" wrapper so it becomes read-only.

REVIEW: ArrayList, LinkedList, HashSet, HashMap

AJ1512.java

```
public class AJ1512 {
    public static void main (String[] args) {

        DyeColor dc;                              //page aj14.07
        CatalogPageAL cp = new CatalogPageAL ();
       //CatalogPageLL cp = new CatalogPageLL ();
       //CatalogPageHS cp = new CatalogPageHS ();
       //CatalogPageHM cp = new CatalogPageHM ();

        //SOCKS
        dc = new DyeColor("RED", 10.10);
        if (cp.add("socks", dc) == false) pErr ("RED");
        dc = new DyeColor("YELLOW", 20.20);
        if (cp.add("socks",dc) == false) pErr ("YELLOW");

        //TIES
        dc = new DyeColor("GREEN", 30.30);
        if (cp.add("ties", dc) == false) pErr ("GREEN");
        dc = new DyeColor("BLUE", 40.40);
        if (cp.add("ties", dc) == false) pErr ("BLUE");

        cp.printData();
    }
    private static void pErr (String s) {
        System.err.println (s + ", add failed, exiting");
        System.exit (1);
    }
}
```

1. DyeColor.java, page aj14.09, is used without changes.

2. On the next four pages, the CatalogPage classes are:

Page	CatalogPage class	collection class
aj15.13	CatalogPageAL.java	ArrayList
aj15.14	CatalogPageLL.java	LinkedList
aj15.15	CatalogPageHS.java	HashSet
aj15.16	CatalogPageHM.java	HashMap

CatalogPageAL.java, ArrayList HAS SEQUENCE, DUPS OK

CatalogPageAL.java

```
import java.util.ArrayList;
import java.util.Iterator;
@SuppressWarnings ( "unchecked" )

public class CatalogPageAL {

    private ArrayList socks = new ArrayList ();
    private ArrayList ties = new ArrayList ();

    public CatalogPageAL () {
    }

    public boolean add (String garment, DyeColor dc){
        if (garment == null || dc == null) {
            return false;
        }
        boolean returnValue = false;
        if (garment.equals("socks") ) {
            socks.add (dc);
            returnValue = true;
        }
        if (garment.equals("ties")) {
            ties.add (dc);
            returnValue = true;
        }
        return returnValue;
    }

    public void printData () {
      //printing via toString() and runtime polymorphism

      for(Iterator i=socks.iterator();i.hasNext();){
          System.out.println ("socks=" + i.next() );
      }
      for (Object o : ties) {
          System.out.println ("ties=" + o);
      }
    }
}
```

Result, AJ1512.java with CatalogPageAL.java

```
socks=DyeColor[RED, 10.1]                       ---entry order
socks=DyeColor[YELLOW, 20.2]                    ---numbers need
ties=DyeColor[GREEN, 30.3]                         formatting
ties=DyeColor[BLUE, 40.4]
```


CatalogPageLL.java, LinkedList HAS SEQUENCE, DUPS OK

<u>**CatalogPageLL.java**</u>

```
import java.util.LinkedList;
import java.util.Iterator;
@SuppressWarnings ( "unchecked" )

public class CatalogPageLL {

    private LinkedList socks = new LinkedList ();
    private LinkedList ties = new LinkedList ();

    public CatalogPageLL () {
    }

    public boolean add (String garment, DyeColor dc){
        if (garment == null || dc == null) {
            return false;
        }
        boolean returnValue = false;
        if (garment.equals("socks") ) {
            socks.add (dc);
            returnValue = true;
        }
        if (garment.equals("ties")) {
            ties.add (dc);
            returnValue = true;
        }
        return returnValue;
    }

    public void printData () {
     DyeColor dc;

     for(Iterator i=socks.iterator();i.hasNext();){
         dc = (DyeColor) i.next();
         System.out.println ("socks=" + dc);
     }
     for (Object o : ties) {
         dc = (DyeColor) o;
         System.out.println ("ties=" + dc);
     }
    }
}
```

<u>**Result, AJ1512.java with CatalogPageLL.java**</u>

```
socks=DyeColor[RED, 10.1]                         ---entry order
socks=DyeColor[YELLOW, 20.2]                      ---numbers need
ties=DyeColor[GREEN, 30.3]                           formatting
ties=DyeColor[BLUE, 40.4]
```


CatalogPageHS.java, HashSet HAS NO SEQUENCE, NO DUPS

CatalogPageHS.java

```
import java.util.HashSet;
import java.util.Iterator;
@SuppressWarnings ( "unchecked" )

public class CatalogPageHS {

    private HashSet socks = new HashSet ();
    private HashSet ties = new HashSet ();

    public CatalogPageHS () {
    }

    public boolean add (String garment, DyeColor dc){
        if (garment == null || dc == null) {
            return false;
        }
        boolean returnValue = false;
        if (garment.equals("socks") ) {
            socks.add (dc);
            returnValue = true;
        }
        if (garment.equals("ties") ) {
            ties.add (dc);
            returnValue = true;
        }
        return returnValue;
    }

    public void printData () {
      DyeColor dc;

      for(Iterator i=socks.iterator(); i.hasNext(); ){
          dc = (DyeColor) i.next();
          System.out.println ("socks=" + dc);
      }
      for(Object o : ties) {
          dc = (DyeColor) o;
          System.out.println ("ties=" + dc);
      }
    }
}
```

Result, AJ1512.java with CatalogPageHS.java

```
socks=DyeColor[YELLOW, 20.2]           ---not in entry order
socks=DyeColor[RED, 10.1]              ---numbers need formatting
ties=DyeColor[BLUE, 40.4]
ties=DyeColor[GREEN, 30.3]
```


CatalogPageHM.java, HashMap HAS KEY-VALUE PAIRS, HASHED ORDER

CatalogPageHM.java

```
import java.util.HashMap;   // ________________________________
import java.util.Set;       // | KEY     | VALUE              |
import java.util.ArrayList; // | String  | ArrayList of       |
                            // | garment | DyeColor objects   |
@SuppressWarnings ( "unchecked" )

public class CatalogPageHM {

    private HashMap hm = new HashMap ();
    ArrayList al = null;

    public CatalogPageHM () {
    }
    public boolean add (String garment, DyeColor dc){
        if (garment == null || dc == null) {
            return false;
        }
        if (hm.containsKey(garment) ) {
            al = (ArrayList) hm.get (garment);
        } else {
            al = new ArrayList ();
        }
        al.add (dc);
        hm.put (garment, al);
        return true;
    }
    public void printData () {
        Set garments = hm.keySet();

        for (Object garmentKey : garments) {
            al = (ArrayList) hm.get (garmentKey);
            System.out.println (garmentKey + "=" + al);

            for (Object dyecolor : al) {
                System.out.println ("    " + dyecolor);
            }
        }
    }
}
```

Result, AJ1512.java with CatalogPageHM.java

```
ties=[DyeColor[GREEN, 30.3], DyeColor[BLUE, 40.4]]
    DyeColor[GREEN, 30.3]
    DyeColor[BLUE, 40.4]
socks=[DyeColor[RED, 10.1], DyeColor[YELLOW, 20.2]]
    DyeColor[RED, 10.1]
    DyeColor[YELLOW, 20.2]
```

WHICH COLLECTION SHOULD YOU USE?

1. To select a collection class, consider these factors:

   a. What methods will be useful?

   b. Do you need synchronization? If so, do you want it built-in in the class you use, or will you wrap an unsynchronized class in a synchronized wrapper?

   c. Are duplicate elements to be accepted?

   d. Should elements be ordered or unordered?

   e. For frequent, fast insertion and removal of elements, Lists, especially LinkedLists, are efficient.

   f. Will you be using an index to your elements? ArrayList is suitable for this.

   g. Do you want your access time to locate an element to be independent of the size of the collection? Hashtable and HashMap are suitable for this.

   h. For random as well as sequential access to elements, use a "balanced tree" such as TreeSet or TreeMap.

2. If future changes in the application may require changing which collections class it uses, if you can limit yourself to using only the methods specified by an interface, you should define your reference to be the interface type. This is called "coding to the interface". Then you can change your collections class to a different class that implements the same interface without changing any method names.

   a. This gives you a tradeoff in flexibility: you limit your methods, but you gain ease in switching from one type of collection to another.

   b. For example, if your reference is type List, then you can change the implementing class from ArrayList to LinkedList and vice versa without changing the reference type.

OPTIONAL: SortedMap and TreeMap, SortedSet and TreeSet

SortedMap, Interface

1. SortedMap is a subinterface of Map, and provides ordering on its keys. The keys must implement Comparable, or a reference to a Comparator for the keys must be passed to the constructor. Iterating over the keys would show the sequence. Methods that use the ordering include firstKey, lastKey, headMap, tailMap, and subMap.

TreeMap implements SortedMap

2. TreeMap implements SortedMap and subinterface NavigableMap, and provides the methods ceilingKey, floorKey, higherKey, lowerKey, headMap, tailMap, and subMap.

SortedSet, Interface

3. SortedSet is a subinterface of Set, and provides ordering on its elements based on their Comparable compareTo method or based on a Comparator whose reference must be passed to the constructor.

TreeSet implements SortedSet

4. TreeSet implements SortedSet as well as Collection, Set, and NavigableSet. Methods that make use of the ordering include first, last, headSet, tailSet, subSet, ceiling, floor, comparator, higher, lower, descendingIterator, descendingSet, pollFirst, and pollLast.

What is a tree?

5. Tree structures are a way of organizing the elements in a collection for efficient adding, ordering, and retrieval of elements. Some kinds of trees are specialized for specific tasks, such as binary search trees used in relational databases and filesystems, hash tables for compilers to look up identifiers, various trees used for internet indexing services, and balanced trees such as used in TreeMap and TreeSet.

6. Using a sequenced TreeMap or TreeSet can be more efficient than repeated sorting of random collection elements. It also reduces the number of lines of code and potential bugs.

AJ1519.java

```
import java.util.Map;
import java.util.TreeMap;
import java.util.List;
import java.util.ArrayList;
public class AJ1519 {
    public static void main(String args[]) {

        Map <String,List<String>> index = new TreeMap<>();

        List<String> listA = new ArrayList<String> ();
        listA.add ("autoboxing");
        listA.add ("abstract");
        List<String> listB = new ArrayList<String> ();
        listB.add ("byte steams");
        listB.add ("boolean");

        index.put ("B", listB);
        index.put ("A", listA);
        System.out.println(index);
    }
}
```

Result, AJ1519.java

```
{A=[autoboxing, abstract], B=[byte steams, boolean]}
```

AJ1519a.java

```
import java.util.Map;
import java.util.TreeMap;
import java.util.Set;
import java.util.TreeSet;
public class AJ1519a {
    public static void main(String args[]) {

        Map <String,Set<String>> index = new TreeMap<>();

        Set<String> setA = new TreeSet<String> ();
        setA.add ("autoboxing");
        setA.add ("abstract");
        Set<String> setB = new TreeSet<String> ();
        setB.add ("byte steams");
        setB.add ("boolean");

        index.put ("B", setB);
        index.put ("A", setA);
        System.out.println(index);
    }
}
```

Result, AJ1519a.java

```
{A=[abstract, autoboxing], B=[boolean, byte steams]}
```


EXERCISES

1. CaseStudy151.java in com.themisinc.u15

    a. Copy CaseStudy141.java and call the copy CaseStudy151.java.

    b. In the main method replace the array of RoomReservation5 objects with an ArrayList of RoomReservation5 objects.

    c. In the main method create a HashMap from the RoomReservation5 elements in the ArrayList. Use the reservation number as key, and the RoomReservation5 object as the value.

    d. Create a method called printBySeats that receives one int parameter that must be 10, 12, or 14, and prints the list of reservation numbers for reservations in the HashMap that have that number of seats. The main method should call printBySeats two times, passing 10 and then 14.

    The output will be similar to that of CaseStudy141.java, but reservation numbers may appear in random sequence.

2. CaseStudy152.java in com.themisinc.u15

    a. Copy CaseStudy151.java and call the copy CaseStudy152.java. Achieve similar results with a different HashMap as follows.

    b. Iterate through the ArrayList and create a key-value pair for each different value that you find in the variable seats in the RoomReservation5 objects. For each seats key, create value consisting of a String with a newline-separated series of reservation numbers.

    c. After you finish creating your HashMap, print the values.

Possible Result for either Case Study

```
Reservations, 10 seats
130325
130322
Reservations, 14 seats
130324
130327
130321
130328
```

SOLUTIONS

CaseStudy151.java in com.themisinc.u15

```
package com.themisinc.u15;  //_______________________________
import java.util.ArrayList; //| KEY     | VALUE              |
import java.util.HashMap;   //| Integer | rr                 |
import java.util.Set;       //|_res num_|_ref to RR5 object_|
@SuppressWarnings ("unchecked")
public class CaseStudy151 {

    private static ArrayList           a = null;
    private static HashMap             h = null;
    private static RoomReservation5   rr = null;
    private static Object       valueRef = null;

    public static void main (String[] args) {

        a = new ArrayList ();
        a.add (new RoomReservation5 (130321, 14, 5, 25.00));
        a.add (new RoomReservation5 (130322, 10, 2, 25.00));
        a.add (new RoomReservation5 (130323, 12, 5, 25.00));
        a.add (new RoomReservation5 (130324, 14, 1, 35.00));
        a.add (new RoomReservation5 (130325, 10, 4, 25.00));
        a.add (new RoomReservation5 (130326, 12, 2, 25.00));
        a.add (new RoomReservation5 (130327, 14, 5, 45.00));
        a.add (new RoomReservation5 (130328, 14, 3, 35.00));

        h = new HashMap ();
        for (Object o : a) {
            if (! (o instanceof RoomReservation5) ) continue;
            rr = (RoomReservation5) o;
            h.put(new Integer(rr.getReservationNumber()),rr);
        }
        printBySeats (10);
        printBySeats (14);
    }

    public static void printBySeats (int sn) { //seats number
        if (sn != 10 && sn != 12 && sn != 14) return;
        System.out.println ("Reservations, "+sn+ " seats");
        Set s = h.keySet();

        for (Object o : s) { //o is Object ref to Integer key
            valueRef = h.get(o);            //valueRef to RR5
            if (valueRef instanceof RoomReservation5) {
                rr = (RoomReservation5) valueRef;
                if (rr.getSeats() == sn) {
                    System.out.println (
                    rr.getReservationNumber());
                }
            }
        }
    }
}
```


CaseStudy152.java in com.themisinc.u15

```
package com.themisinc.u15;  //_______________________________
import java.util.ArrayList; //| KEY        | VALUE              |
import java.util.HashMap;   //| Integer    | String             |
                            //|_10 or 14_|_130322\n130325\n_|
@SuppressWarnings ("unchecked")

public class CaseStudy152 {
    public static void main (String[] args) {

        ArrayList a = new ArrayList ();
        a.add (new RoomReservation5 (130321, 14, 5, 25.00));
~~~~~~~~~
        a.add (new RoomReservation5 (130328, 14, 3, 35.00));

        HashMap h = new HashMap ();         //Each Integer key
        h.put(10, "");   //autobox 10       //has zero-length
        h.put(14, "");   //autobox 14       //String as value

        StringBuilder sb = new StringBuilder ();
        int seats = 0;

        for (Object o : a) {
            if(!(o instanceof RoomReservation5)) {continue;}
            RoomReservation5 rr = (RoomReservation5)o;

/*1*/       sb.delete(0,sb.length());    //empty the sb
/*2*/       seats = rr.getSeats();       //seats is an int
/*3*/       if (seats == 10 || seats == 14){

/*4*/           //Get existing String of res nums for seats
                //sb.append ((String)h.get(seats)); //autobox
                Integer seatsKey = new Integer (seats);
                sb.append ( (String) h.get(seatsKey) );

/*5*/           //append new res num to end of String
                sb.append ( rr.getReservationNumber() );

/*6*/           //append \n delimiter char after each res num
                sb.append ( '\n' );

/*7*/           h.put (seatsKey, sb.toString() );
            }
        }
        System.out.println ("Reservations with 10 seats");
        System.out.println (h.get(10)); //autobox 10

        System.out.println ("Reservations with 14 seats");
        System.out.println (h.get(14)); //autobox 14
    }
}
```


COLLECTIONS REVIEW CHART

Class	Number of elements can change after con-struction?	Thread-Safe?	What is the element type?	Object refs can point to any object type?	Are the elements key-value pairs?	Sequence and use of indexes, or Random?	Imple-mented interface
arrays	No	No	any ref or basic type	Yes	No	Sequence	none
Vector	Yes	Yes	Object refs	"	No	Sequence	Collection, List
Stack	"	Yes	"	"	No	Sequence	Collection, List
ArrayList	"	No	"	"	No	Sequence	Collection, List
LinkedList	"	No	"	"	No	Sequence	Collection, List, Deque, Queue
Hashtable	"	Yes	"	"	Yes	Random	Map
HashMap	"	No	"	"	Yes	Random	Map
HashSet	"	No	"	"	Keys only	Random	Collection, Set

Interface	Overview
Collection	No class implements Collection directly. Collection refs can point to various collections. Does not have sequence. Allows duplicates.
List	Extends Collection. Has sequence. Allows insertion at an index position. Indexes start at zero. Typically allows duplicates and multiple null elements.
Iterator	Iterate forward only. Can remove elements.
ListIterator	Extends Iterator. Iterate forward or backward. Can add, replace, or remove elements.
Set	Extends Collection. Allows no duplicates and no sequence.
Map	For key-value pairs, aka hash tables. Does NOT extend Collection. Uses different method names.

(blank)

UNIT 16: GENERICS AND JAVA 5 COLLECTIONS

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

1. Briefly describe the purpose of generics.

2. Use generics to make programs more reliable during execution by enabling the compiler to detect errors in class types.

PROBLEMS WITH LEGACY COLLECTIONS

1. The legacy collections of the Java 2 Collections Framework treat all elements, keys, and values as Objects (use Object type references to point to them). This enables a legacy collection to contain objects of any class type.

2. The class type of a reference determines what identifiers are in scope and can be accessed within the pointed-to object. Using Object references limits the identifiers in scope to those defined in the Object class.

   a. To access identifiers defined in the actual class type of an element, key, or value, you have to determine the actual class type of the object via instanceof, and then cast the reference to its actual class type.

   b. Errors in casting cannot be detected by the compiler, so they cause runtime exceptions.

   c. Use of instanceof and casting can make code cumbersome to read and maintain.

3. Java 5 introduced classes, interfaces, and Enums that use generics to reduce the need for instanceof and casting, and to enable the compiler to verify that types are correct.

4. Starting with Java 5, compilers check for use of generics. If a class, interface, or Enum that is defined with generics is not coded with generics, javac prints warnings and does not compile. To avoid such warnings, specify the annotation @SuppressWarnings ("unchecked").

<u>Result, AJ1603.java without @SuppressWarnings ("unchecked")</u>
```
Note: AJ1603.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

<u>Result from commandline   javac -Xlint:unchecked AJ1603.java</u>
```
AJ1603.java:9: warning: [unchecked] unchecked call to add(E) as a
member of the raw type java.util.ArrayList
        a.add ("string");
              ^
AJ1603.java:10: warning: [unchecked] unchecked call to add(E) as
a member of the raw type java.util.ArrayList
        a.add (74);
              ^
2 warnings
```

CODE COMPLEXITY WITH DATA TYPES AND CASTING, EXAMPLE

AJ1603.java

```
import java.util.ArrayList;
@SuppressWarnings ("unchecked")
public class AJ1603 {

    public static void main (String [] args) {

        ArrayList a = new ArrayList ();
        a.add ("my String data");
        a.add (16.03);  //autoboxed to Double

        printAll (a);
    }

    private static void printAll (ArrayList aList) {
        String sRef = null;
        Double dRef = null;
        char c;
        int i;

        for (Object o : aList) {
            System.out.println ("1. o=" + o);  //Runtime
                                               //polymorphism
            if (o instanceof String) {
                sRef = (String) o;
                c = sRef.charAt(0);  //method in String class
                System.out.println ("2. c=" + c);
            }

            if (o instanceof Double) {
                dRef = (Double) o;
                i = dRef.intValue(); //method in Double class
                System.out.println ("3. i=" + i);
            }
        }
    }
}
```

Result, AJ1603.java with @SuppressWarnings ("unchecked")

```
1. o=my String data
2. c=m
1. o=16.03
3. i=16
```


JAVA 5 COLLECTIONS FRAMEWORK AND GENERICS

1. Java 5 introduced new collections interfaces and classes, such as Queue<E> and PriorityQueue<E>. Existing interfaces and classes were modified by adding generics. Some examples:

Java 2 Interfaces	Java 5 Interfaces
Collection	Collection<E>
List	List<E>
Set	Set<E>
Map	Map<K,V>
Iterator	Iterator<E>
ListIterator	ListIterator<E>

Java 2 Classes	Java 5 Classes
Vector	Vector<E>
ArrayList	ArrayList<E>
LinkedList	LinkedList<E>
Hashtable	Hashtable<E>
HashSet	HashSet<E>
HashMap	HashMap<K,V>

2. The notation <E> creates a "type parameter" called E. Any identifier can be used as the name of a type parameter. By convention, to make type parameters look different from ordinary variables, the names are single uppercase letters. Names commonly used in the Java API are:

name	meaning
E	Element, used in the Collections Framework
K	Key
N	Number
T	Type
V	Value
S, U, etc.	additional types, such as <S>, <U>, etc.

3. Classes and interfaces with "generic type declarations" such as public class HoldVar<T> are also known as "parameterized types."

4. When the word "type" is used in talking about generics, it means a class or interface type.

5. The terms "generic class", "generic type", and "parameterized type" mean a class or interface with generic type parameters.

6. The term "generic type invocation" means calling a method when generic types are involved.

GENERICS, TERMINOLOGY AND SYNTAX

HoldVar.java

```
1   public class HoldVar<T> {                           // parameter T
2       private T var;                          // T instead of Object
3       public HoldVar (T var) {                // T instead of Object
4           setVar (var);
5       }
6       public T getVar () {                    // T instead of Object
7           return var;
8       }
9       public void setVar (T var) {            // T instead of Object
10          this.var = var;
11      }
12      public void printVar () {
13          System.out.println ("printVar=" + var);
14      }
15  }
```

===

1. Line 1 is a "generic type declaration."

2. The notation <T> introduces a "type variable" called T that is associated with the HoldVar class. Type variables are also called "formal type parameters" or "type parameters."

3. After the type variable T is introduced, it can be used anywhere in the class to specify the type.

4. The purpose of a type variable is to allow a type to be specified at compile time. The specified type can be any class or interface type, or another type variable. It can NOT be a basic data type.

5. When a class has multiple type parameters, each parameter letter must be different. HoldVar<T,T> is an error.

6. The terminology for generics resembles that for arrays.

```
Integer[] a           //a references an "array of Integers"
HoldVar<Integer> h    //h references a "HoldVar of Integers"
```

7. Type variables are placeholders for types that you will specify elsewhere. Type variables are not themselves class or interface types.

GENERICS, NOTES

1. Generic parameters may be called "abstract data types," and are said to be replaced by "actual types" during compilation to eliminate the need for "downcasting".

2. Restrictions on generics:

   a. No use of basic data types, so use wrapper classes.

   b. Static members of classes cannot use generic types.

   c. Disallow use of subtypes of the generic type.

      1)
      ```
      HoldVar<Number> hN = new HoldVar<Number> (5);
      HoldVar<Integer>  hI = hN;
      //Type mismatch: cannot convert from HoldVar<Number>
      //to HoldVar<Integer>
      ```

      2) This is ok:
      ```
      HoldVar<Number> hN = new HoldVar<Number> (5);
      hI.setVar (new Integer (22) );
      ```

   d. Disallow declaration of arrays of generic types.

      1) If T is a known generic symbol, you can create an array reference with type T, but you cannot delare an array of generic type.

      ```
      private T[] arrayRef; //compiles
      arrayRef = new T[5];  //error: generic array creation
      ```

3. Benefits of generics:

   a. Compiler can catch errors that previously created runtime problems.
   b. Most downcasting is not required.
   c. A single class can handle multiple parameter types, enabling projects to avoid "code bloat."

4. Starting in Java 7, calls to the constructor of a generic class can use the shorthand of empty angle brackets <> called the diamond operator. The compiler infers the type arguments from the context. As of Java 7, these lines are equivalent:

   1)
   ```
   HoldVar<Integer> h = new HoldVar<Integer> (var);
   ```
   2)
   ```
   HoldVar<Integer> h = new HoldVar<> (var);
   ```


ERROR IN TYPE DETECTED BY COMPILER, EXAMPLE

AJ1607.java

```
1   public class AJ1607 {
2       public static void main (String [] args) {
3
4           Integer seven = new Integer (7);
5
6           HoldVar<Integer> h = new HoldVar<Integer> (seven);
7
8           Integer ref = h.getVar();      //Integer is returned
9           h.printVar();
10
11          h.setVar (new String("16"));   //causes compile error
12      }
13  }
```

HoldVar.java

```
1   public class HoldVar<T> {                          // parameter T
2       private T var;                             // T instead of Object
3       public HoldVar (T var) {                   // T instead of Object
4           setVar (var);
5       }
6       public T getVar () {                       // T instead of Object
7           return var;
8       }
9       public void setVar (T var) {               // T instead of Object
10          this.var = var;
11      }
12      public void printVar () {
13          System.out.println ("printVar=" + var);
14      }
15  }
```

Result of compiling AJ1607 in

```
Exception in thread "main" java.lang.Error: Unresolved
compilation problem:
     The method setVar(Integer) in the type HoldVar<Integer> is
not applicable for the arguments (String)

     at AJ1615.main(AJ1615.java:11)
```

Result of compiling AJ1607.java on a commandline

```
AJ1607.java:10: setVar(java.lang.Integer) in HoldVar<java.lang.In
teger> cannot be applied to (java.lang.String)
        hv.setVar (new String("16"));
           ^
1 error
```


ArrayList<E>, HashSet<E>

AJ1608.java

```
1   import java.util.ArrayList;
2   import java.util.HashSet;
3   import java.util.Collection;
4   import java.util.Iterator;
5   import java.util.List;
6   import java.util.Set;
7   public class AJ1608 {
8
9       private static List<Integer> a =
10          new ArrayList<Integer> ();
11      private static Set<Integer> h =
12          new HashSet<Integer> ();
13
14      public static void main (String [] args) {
15          a.add (new Integer(1) );
16          a.add (2);
17        //a.add ("three");                      //compile error
18          printAll (a);
19
20          h.add (new Integer(4) );
21          h.add (5);
22        //h.add ("six");                        //compile error
23          printAll (h);
24      }
25      private static void printAll (Collection<Integer> c) {
26          Iterator<Integer> it = c.iterator();
27
28          while (it.hasNext() ) {
29              Integer i = it.next();    //no instanceof or cast
30              System.out.print (i + ", ");
31          }
32      }
33  }
```

Result, AJ1608.java

```
1, 2, 4, 5,
```

===

1. Above, use of generics enables the compiler to check that only Integer type objects are added to the two collections. Lines that would cause compiler errors are commented out.

2. Regardless of generics, a Collection still contains Object references, but the compiler ensures that the object type is correct. Thus line 29 is allowed with no instanceof or casting.

LinkedList<E>

AJ1609.java

```
1   public class AJ1609 {                                 //see aj14.11
2       public static void main (String[] args) { //and aj14.24
3
4           LIFO1609 <String> stack = new LIFO1609 <String> ();
5
6           stack.pushLifo ("zero");
7           stack.pushLifo ("one");
8         //stack.pushLifo (new Integer(2) );       //compile error
9
10          while ( ! stack.isEmpty() )
11              System.out.println ("pop=" + stack.popLifo());
12      }
13  }
```

LIFO1609.java

```
import java.util.LinkedList;

public class LIFO1609<T> {

    private LinkedList<T> lifo = new LinkedList<T> ();

    public void pushLifo (T t) {  //only type T can be pushed
        lifo.addLast (t);
    }
    public T popLifo () {                  //Not a generic method!
        T t = null;                        //T is the return type.
        if (lifo.size() != 0) {
           t = lifo.getLast();
           lifo.removeLast();
        }
        return t;
    }
    public boolean isEmpty () {
        if (lifo.size() == 0)
            return true;
        else
           return false;
    }
}
```

Result, AJ1609.java

```
pop=one
pop=zero
```

===

1. Line 11 in main causes runtime polymorphism because an Object reference points to a String object (parent reference to child object), and an overriding method is called (the String class's toString method overrides Object's toString method).

HashMap<K,V>

AJ1610.java

```
import java.util.Map;                                   //see aj15.07
import java.util.HashMap;
public class AJ1610 {
   public static void main (String [] args) {

      String [] phoneNumbers = {
         "914-456-1234",
         "212-123-1234",
         "415-778-1234",
         "914-724-1234",
         "415-224-1234"
      };

      Map<String,Integer> m = new HashMap<String,Integer> ();

      Integer oldHowMany;
      int newHowMany;

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

         String areaCode = phoneNumbers[i].substring(0,3);

         if ( m.containsKey(areaCode) ) {

            oldHowMany = m.get(areaCode);//no cast needed
            newHowMany = 1 + oldHowMany;       //unboxing

         } else {

            newHowMany = 1;
         }

         m.put(areaCode, newHowMany);         //autobox int
      }                                       //to Integer
      System.out.println (m);
   }
}
```

Result, AJ1610.java

```
{212=1, 415=2, 914=2}
```


OPTIONAL: Map.Entry<K,V>

AJ1611.java

```java
import java.util.HashMap;
import java.util.Set;
import java.util.Map;
import java.util.Iterator;

public class AJ1611 {
    public static void main (String[] args) {

        HashMap <String,Integer> h =
            new HashMap <String,Integer> ();

        h.put ("one",   new Integer(1) );
        h.put ("two",   new Integer(2) );
        h.put ("three", new Integer(3) );

        Set <Map.Entry<String,Integer>> kv = h.entrySet();

        Iterator <Map.Entry<String,Integer>> i =
            kv.iterator();

        while (i.hasNext()) {
            Map.Entry me = i.next();
            System.out.println (
                me.getKey() + "=" + me.getValue() );
        }
    }
}
```

Result, AJ1611.java

```
two=2
one=1
three=3
```

===

1. Map.Entry<K,V> is an interface with these methods:

   a. boolean equals(Object o); Compare o with this entry.
   b. K getKey(); Return key for this entry.
   c. V getValue(); Return value for this entry.
   d. int hashCode(); Return hash code for this entry.
   e. V setValue(V val); Replace this entry's value with val.

OPTIONAL: GENERIC METHODS

1. A generic methods are not frequently used. A generic method is a method with one or more type parameters defined before (to the left of) the method's return type.

```
public static <T> int countElem (T[] a, T countMe) {
```

2. Static and non-static methods can have type parameters.

3. On the facing page, the method countStr is not generic, and counts the number of times a given String occurs in a String array.

4. On the facing page, the method countElem is generic, and can work with arrays of any class type.

   a. Type parameter <T> is defined before the return type in the method header. <T> makes the method generic and specifies the name of the type parameter.

   b. Type parameter T is used in the parameter list and method body.

   c. When the generic method is called, the type for the type parameter is not explicitly coded. The compiler infers the type from the arguments in the method call. Here, the compiler can infer the type of T from the method parameters a and countMe.

5. Because a basic type int is passed to countElem, the compiler autoboxes the int value to an Integer, which is the appropriate wrapper class. The following two statements are equivalent.

```
int elem21 = countElem (iArray, 21);            //autobox 21
int elem21 = countElem (iArray, new Integer(21));
```

6. Generics can not use basic types, so countElem cannot count how many times an int occurs in an int array.

OPTIONAL: GENERIC METHOD, EXAMPLE

AJ1613.java

```
public class AJ1613 {
    public static void main (String[] args) {

        String[]  sArray = {"A", "B", "C", "B", "D"};
        Integer[] iArray = {21, 34, 65, 21, 987, 21};

        int strB   = countStr  (sArray, "B");
        int elemB  = countElem (sArray, "B");
        int elem21 = countElem (iArray, 21);   //autobox 21

        System.out.println(
              "strB="   + strB +
            ", elemB="  + elemB +
            ", elem21=" + elem21);
    }
    public static int countStr (String[] a, String countMe){
        int total = 0;

        if (countMe == null) {
            for (String elem : a) {
                if (elem == null) total++;            //nulls
            }
       } else {
            for (String elem : a) {
                if (countMe.equals(elem)) total++;//non-nulls
            }
       }
       return total;
    }
    public static <T> int countElem (T[] a, T countMe) {
        int total = 0;

        if (countMe == null) {
            for (T elem : a) {
                if (elem == null) total++;            //nulls
            }
        } else {
            for (T elem : a) {
                if (countMe.equals(elem)) total++;//non-nulls
            }
        }
        return total;
    }
}
```

Result, AJ1613.java

```
strB=2, elemB=2, elem21=3
```


OPTIONAL: GENERIC CONSTRUCTORS

AJ1614a.java

```
public class AJ1614a {                              //non-generic class

     public static void main (String[] args) {
          AJ1614a ref1 = new AJ1614a ("hello");
          AJ1614a ref2 = new AJ1614a (new Double(1.2));
     }

     public <U> AJ1614a (U u) {
         String s = u.toString();
         System.out.println ("s=" + s);
     }
}
```

Result, AJ1614a.java

```
s=hello
s=1.2
```

AJ1614b.java

```
public class AJ1614b <T> {                              //generic class

     public static void main (String[] args) {
          AJ1614b <Long> refL = new AJ1614b <Long> ("gen");
          AJ1614b <String> refS = new AJ1614b <String> (1.2);
     }

     public <U> AJ1614b (U u) {       //class is T, param is U
          String s = u.toString();
          System.out.println ("U=" + u.getClass() );
          System.out.println ("s=" + s);
     }
}
```

Result, AJ1614b.java

```
U=class java.lang.String
s=gen
U=class java.lang.Double
s=1.2
```

===

1. If AJ1614b's constructor header were public AJ1614b(T param) then the parameter would have to be the same type specified for type T.

TYPE ERASURE

AJ1615.java
```
import java.util.ArrayList;
public class AJ1615 {
    public static void main (String[] args) {

        ArrayList<String> a = new ArrayList<String> ();
        System.out.println ( a.getClass() );
    }
}
```

Result, AJ1615.java
```
class java.util.ArrayList
```

==

1. Type variables are not part of the class name. Type variables are deleted by the compiler after it uses the generic types to check type safety (correct use of types) and detect errors. Thus, objects created for generic types do not contain information about type parameters. This is called type erasure.

2. The purpose of type erasure is to ensure binary compatibility between Java classes and applications, regardless whether they were created with or without generics, so new code can work with legacy code that uses "raw types."

3. The getClass() instance method is inherited by every class from Object.

4. Due to type erasure, the following operations will not work because the type of T would have been erased:

```
public class AJ1615x<T> {
    public static void erasureMethod (Object o) {
        if (o instanceof T) { }  //Compiler error
        T r1 = new T();          //Compiler error
        T[] tArray = new T[4];   //Compiler error
        T r2 = (T) new Object(); //Unchecked cast warning
    }
}
```


BOUNDED TYPE PARAMETERS, EXTENDS

HoldNum.java

```
public class HoldNum<T extends Number> {    //can use only
    private T num;                          //Number or
    public HoldNum (T num) {                //subclasses of
        setNum (num);                       //Number
    }
    public T getNum () {
        return num;
    }
    public void setNum (T num) {
        this.num = num;
    }
}
```

AJ1616.java

```
public class AJ1616 {
    public static void main (String[] args) {

      //HoldNum<String>  hS = new HoldNum<String> ("ab");

        HoldNum<Number>  hN = new HoldNum<Number>(1);
        HoldNum<Integer> hI = new HoldNum<Integer>(2);
        HoldNum<Double>  hD = new HoldNum<Double> (3.4);

        System.out.println(hN + ", " + hI + ", " + hD);
        System.out.println(hN.getNum()
            + ", " +  hI.getNum() + ", " + hD.getNum());
    }
}
```

Result, AJ1616.java

```
HoldNum@1db9742, HoldNum@106d69c, HoldNum@52e922
1, 2, 3.4
```

==

1. "Bounded type parameters" enable you to restrict the types allowed to be passed for a type parameter, so the compiler will allow only a specified class and its subclasses.

2. A bounded type parameter is declared by a type name followed by "extends" followed by the class which is the upper bound. When used like this, "extends" means "extends a class or implements an interface."

3. To specify additional interfaces that must be implemented, use the & character:

```
<U extends MySuperclass & MyInterface1 & MyInterface2>
```

4. In the example above, HoldNum<T extends Number> allows only references of Number or subclasses of Number to be specified.

BOUNDED TYPE PARAMETERS, EXTENDS VERSUS SUPER

AJ1617.java

```
1   import java.util.ArrayList;
2   import java.util.List;
3   public class AJ1617 {
4       public static void main (String[] args) {
5
6   /*1*/   ArrayList<String> alStr = new ArrayList<String> ();
7               alStr.add("a");
8           ArrayList<Integer> alI = new ArrayList<Integer> ();
9                alI.add(1);
10          ArrayList<Double> alD = new ArrayList<Double> ();
11              alD.add(2.3);
12
13  /*2*/   List<? extends Number> listE = alI; //can't use alStr
14          printE (listE);
15
16          listE = alD;
17          printE (listE);
18
19  /*3*/   ArrayList<Number> alN = new ArrayList<Number> ();
20              alN.add(1);
21              alN.add(2.3);
22
23          List<? super Integer> listS = alN;
24          printS (listS);
25      }
26      public static void printE (List<? extends Number> x ) {
27          for (Number n : x){System.out.print(n + ",  "); }
28      }
29      public static void printS (List<? super Integer> x) {
30          for (Object o : x){System.out.print(o + ",  "); }
31      }
32  }
```

Result, AJ1617.java

```
1,  2.3,  1,  2.3,
```

===

1. Bounded wildcards use extends to specify the type of the upper bound. The specified type must be the same class, a subtype of the class, or an implementing class.

2. Bounded wildcards use super to specify that the type must be a supertype of the bound. Note that on line 30, reference type Number is not accepted, and Object must be specified.

CAUTION WITH PARAMETER SUBTYPES AND SUPERTYPES

AJ1618.java

```
1   public class AJ1618 {
2       public static void main (String [] args) {
3           HoldVar<Number> n = new HoldVar<Number> (12);
4           p (n);
5           HoldVar<Integer> i = new HoldVar<Integer> (34);
6           p (i);
7
8           //n = i;
9       }
10      private static void p (HoldVar<? extends Number> h){
11          System.out.println ("var=" + h.getVar() );
12      }
13  }
14  class HoldVar<T> {
15      private T var;
16      public HoldVar (T var) {
17          setVar (var);
18      }
19      public T getVar () {
20          return var;
21      }
22      public void setVar (T var) {
23          this.var = var;
24      }
25      public void printVar () {
26          System.out.println ("printVar=" + var);
27      }
28  }
```

Result, AJ1618.java

```
var=12
var=34
```

===

1. Generic classes instantiated using different parameter types are treated as different data types by the compiler.

2. The compiler does not allow references to generic subclasses and generic superclasses to be assigned to each other. When line 8 above tries to assign supergenerictype reference n to point to a HoldVar<Integer> you get:

```
Exception in thread "main" java.lang.Error: Unresolved
compilation problem:
        Type mismatch: cannot convert from HoldVar<Integer>
        to HoldVar<Number>

        at AJ1618.main(AJ1618.java:8)
```


EXERCISES

1. Create E161.java by modifying E161StarterCode.java below to use generics so that only Strings can be stored in the two Collections. The compiler should not display any warnings. Lines that create compiler errors should be modified or commented out.

```
package com.themisinc.u16;
import java.util.Collection;
import java.util.HashSet;
import java.util.Iterator;
import java.util.ArrayList;
public class E161StarterCode {
    public static void main (String [] args) {
        Collection c = new HashSet ();
            c.add ("1");
            c.add (2);
        System.out.println ("HashSet elements:" + c);
        Iterator it = c.iterator();
        while (it.hasNext() )
            System.out.print ("H=" + it.next() + "   ");
        System.out.println ();

        c = new ArrayList ();
            c.add ("3");
            c.add (4);
        System.out.println ("ArrayList elements:" + c);
        it = c.iterator();
        while (it.hasNext() )
            System.out.print ("A=" + it.next() + "   " );
        System.out.println ();
    }
}
```

2. Create CaseStudy162.java that has a local ArrayList in the main method. Copy the add statements from CaseStudy151.java. Do not use @SuppressWarnings. Use generics to tie the ArrayList to RoomReservation5 type. Call a printBySeats method and pass an int of 10 or 14 and the reference to your ArrayList. The method should print the reservation numbers for reservations with the parameter number of seats.

3. Copy CaseStudy162.java and call the copy CaseStudy163.java. Create a separate class to contain the method printBySeats, and achieve the same printout.

SOLUTIONS

E161.java in com.themisinc.u16

```
package com.themisinc.u16;
import java.util.Collection;
import java.util.HashSet;
import java.util.Iterator;
import java.util.ArrayList;
public class E161 {
    public static void main (String [] args) {
        Collection<String> c = new HashSet<String> ();
            c.add ("1");
            c.add ("2");                           //modified
        System.out.println ("HashSet elements:" + c);
        Iterator<String> it = c.iterator();
        while (it.hasNext() )
            System.out.print ("H=" + it.next() + "   ");
        System.out.println ();

        c = new ArrayList<String> ();//has to be <String>
            c.add ("3");           //because c is <String>
            c.add ("4");                           //modified
        System.out.println ("ArrayList elements:" + c);
        it = c.iterator();
        while (it.hasNext() )
            System.out.print ("A=" + it.next() + "   " );
        System.out.println ();
    }
}
```

Result, E161.java in com.themisinc.u16

```
HashSet elements:[2, 1]
H=2   H=1
ArrayList elements:[3, 4]
A=3   A=4
```

<u>CaseStudy162.java in com.themisinc.u16</u>

```
package com.themisinc.u16;
import java.util.ArrayList;
//@SuppressWarnings ("unchecked")

public class CaseStudy162 {
    public static void main (String[] args) {

        ArrayList<RoomReservation5> a =
            new ArrayList<RoomReservation5> ();

        a.add (new RoomReservation5 (130321, 14, 5, 25.00));
        a.add (new RoomReservation5 (130322, 10, 2, 25.00));

        a.add (new RoomReservation5 (130323, 12, 5, 25.00));
        a.add (new RoomReservation5 (130324, 14, 1, 35.00));

        a.add (new RoomReservation5 (130325, 10, 4, 25.00));
        a.add (new RoomReservation5 (130326, 12, 2, 25.00));

        a.add (new RoomReservation5 (130327, 14, 5, 45.00));
        a.add (new RoomReservation5 (130328, 14, 3, 35.00));

        printBySeats (10, a);
        printBySeats (14, a);
    }

    public static void printBySeats (int sn,   //seats number
        ArrayList<RoomReservation5> a) {

        if (sn!=10 && sn!=12 && sn!=14) return;

        System.out.println ("Reservations with " +
            sn + " seats");

        for (RoomReservation5 rr : a) { //due to generics you
            if (rr.getSeats() == sn) {  //don't need to cast
              System.out.println (rr.getReservationNumber());
            }
        }
    }
}
```

<u>Result, CaseStudy162.java in com.themisinc.u16</u>

```
Reservations with 10 seats
130322
130325
Reservations with 14 seats
130321
130324
130327
130328
```


CaseStudy163.java in com.themisinc.u16

```
package com.themisinc.u16;
import java.util.ArrayList;
public class CaseStudy163 {
    public static void main (String[] args) {
        ArrayList<RoomReservation5> a =
            new ArrayList<RoomReservation5> ();
        a.add (new RoomReservation5 (130321, 14, 5, 25.00));
        a.add (new RoomReservation5 (130322, 10, 2, 25.00));
       a.add (new RoomReservation5 (130323, 12, 5, 25.00));
        a.add (new RoomReservation5 (130324, 14, 1, 35.00));
        a.add (new RoomReservation5 (130325, 10, 4, 25.00));
        a.add (new RoomReservation5 (130326, 12, 2, 25.00));
        a.add (new RoomReservation5 (130327, 14, 5, 45.00));
        a.add (new RoomReservation5 (130328, 14, 3, 35.00));

        Printer163 p = new Printer163 (a);
        p.printBySeats (10);
        p.printBySeats (14);
    }
}
```

Printer163.java in com.themisinc.u16

```
package com.themisinc.u16;
import java.util.ArrayList;
public class Printer163 {

    private ArrayList<RoomReservation5> a = null;

    public Printer163 (ArrayList<RoomReservation5> a) {
        this.a = a;
    }

    public void printBySeats (int sn) { //seats number
        if (sn!=10 && sn!=12 && sn!=14) return;

        System.out.println ("Reservations with " +
            sn + " seats");

        for (RoomReservation5 rr : a) {     //no instanceof
            if (rr.getSeats() == sn) {      //or cast needed
              System.out.println (rr.getReservationNumber());
            }
        }
    }
}
```

Result, CaseStudy163.java in com.themisinc.u16

```
Reservations with 10 seats
130322
130325
Reservations with 14 seats
130321
130324
130327
130328
```


UNIT 17: UNIT TESTING WITH JUnit

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

1. Briefly describe unit testing, why it is important, and how to do it.

2. Create and execute a JUnit TestCase class, and review the results.

3. Create and execute a JUnit TestSuite class, and review the results.

WHAT IS UNIT TESTING? WHAT IS JUnit?

1. Unit testing is a modular way to test modular code.

    a. Java applications are modular, typically consisting of one or more classes, each with a public interface.

    b. Typically, unit testing for Java code is done class by class, and tests each method to prove that it works correctly.

2. JUnit is a unit testing tool that was written in Java to work on multiple platforms.

    a. JUnit, introduced in 1998, is one of a family of unit testing frameworks known as xUnit.

    b. JUnit was developed by Kent Beck, Erich Gamma, David Saff, and Mike Clark of the University of Calgary.

    c. JUnit provides a simple framework for developing and executing repeatable, standardized tests for Java code.

        1) JUnit facilitates describing how a method is expected to work.

        2) Tests automatically report their results using a GUI.

    d. Some other kinds of testing are:
        1) Integration testing, to show that two or more modules work together correctly.
        2) System testing, to show that all components work together correctly.
        3) User acceptance testing, to determine whether the software system works as the user wants.

3. Two releases of JUnit are in current use. Release 4 uses annotations. Release 3 does not.

4. JUnit is automatically included in your download of Eclipse. To download JUnit separately, go to http://junit.org/

5. Documentation for the classes and annotations of JUnit 4 is at http://junit.sourceforge.net/javadoc/

DESIGNING GOOD TESTS

1. For each business class, create one test class.

2. For each method in the business class, if the method does significant work, create a series of test methods for it.

3. Each test method should be small and focus on one testable consideration. Usually many test methods are needed for each business method. You should make a separate test for:

   a. Each path through the code.

   b. Each positive outcome (code works) and negative outcome (code fails).

      1) "Positive tests" show that code works as it should when things go well

      2) "Negative tests" show that code fails as it should when things go wrong

   c. Each possible kind of good and bad data.

   d. Upper and lower boundaries, such as the length of a String or array, and the value of a numeric variable.

      1) Minimum
      2) One less that minimum
      3) Maximum
      4) One more than maximum
      5) A value in the middle of the valid range

4. A get method may not need to be separately tested, if it is fully tested when it returns the result of the corresponding set method.

5. After the test method calls the business method:

   a. Did a set method assign the value correctly?

   b. Are the contents of an array or collection correct after execution of the business method?

TEST-DRIVEN DEVELOPMENT: CODE A LITTLE, TEST A LITTLE

1. JUnit facilitates test-driven development: code a little, test a little.

2. The process of test-driven development is:

   a. Create a business class with stubs for its methods.

   b. Create a test class for the business class. This is called an "unimplemented test". In Eclipse each test method will contain

      ```
      fail("Not yet implemented");
      ```

   c. Code test methods in the test class. Each test method calls a business method and compares expected to actual results.

   d. Add code to implement each business method.

   e. Run the test class as a JUnit test to test each business method.

3. If several business classes work together, their test classes can be put together into a test suite. Test suites are covered later in this unit.

ADVANTAGES

1. Advantages of unit testing:

    a. Facilitates simple, planned, documented, thorough, automated, and standardized testing of business code.

    b. Helps to improve code quality.

    c. Helps to find bugs early.

    d. Increases developer confidence in code.

    e. Increases developer productivity.

    f. Facilitates future code maintenance and testing, and makes it more likely that thorough tests will be run.

    g. Can assist in refactoring code, and can provide the rationale for refactoring if code is difficult to test.

        1) Refactoring is the restructuring of a unit's code to improve its quality, without changing its external behavior. A book on refactoring by Martin Fowler is *Refactoring: Improving the Design of Existing Code*.

2. Advantages of automated testing:

    a. Shows the location of code problems via a graphical interface (JUnit's GUI is easy to learn and use).

    b. Simplifies comparison of expected versus test results.

    c. Eliminates the need to manually and repeatedly enter test data, capture results, and handle errors.

    d. Facilitates reuse of tests after fixing bugs.

    e. Simplifies regression testing (retesting of modified software to ensure that any bugs were fixed, previously working code still works, and newly-added code has not introduced errors, aka verification testing).

3. Unit testing is worth the effort to set up, despite reasons commonly given for not doing it:

    a. "We don't have time for it."
    b. "We don't have the budget."
    c. "The manager doesn't think it's worth the effort."
    d. "Writing unit tests is boring. I'd rather be developing code."
    e. "Let others take care of testing."

GUIDELINES

1. Plan for unit testing during project inception.

2. Develop a written test plan that documents tests and expected inputs and results.

3. Keep the tests simple to encourage use and reuse, save time, and reduce resource requirements.

4. Invite every team member to assist and to comment on the test plan, in order to create comprehensive testing.

5. Support regression testing via test case updates, version control that is in sync with version control of the software, and documentation, so that new code or modified code can be easily fully tested.

6. Standardize the form of unit test plans and documentation for use by all team members.

BEST PRACTICES

1. Minimize repeated code in the test methods by using test fixtures, and using the setUp method to assign them. Test fixtures and the setup method are covered later in this unit.

2. Each test class should be able to be run independently, and should not depend on the results of another test class.

3. Each test method should be independent, and should not depend on the results of another test method.

    a. Dependencies make test code very hard to work with.

    b. The effect of dependencies can be unpredictable because <u>the order in which test methods are executed is unpredictable</u>.

4. Maintain test classes so they stay in sync with the business code as requirements change.

5. Use the same version control for test classes and business classes.

6. Automate as much as possible.

7. Regression test entire classes or sets of related classes after significant code changes.

AJ1708.java, AJ1708Test.java

AJ1708.java

```
package com.themisinc.u17;
public class AJ1708 {

    private String name;

    public AJ1708 (String name) {
        setName (name);
    }

    public int setName (String name) {
        if (name==null) return -1;
        if (name.length() > 0 && name.length() < 11) {
            this.name = name;
            return 1;
        }
        return -1;
    }
    public String getName () {
        return name;
    }
}
```

AJ1708Test.java

```
package com.themisinc.u17Test;
import com.themisinc.u17.AJ1708;

import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;              //static import

public class AJ1708Test {

    private AJ1708 aj;                  //declare test fixture

    @Before      //initialize fixture before each test method
    public void setUp() throws Exception {
         aj = new AJ1708 ("Unit Test");
    }

    @Test       //subject under test_scenario_expected result
    public void testSetName_NullValue_bad() {

        long returnFromSet = aj.setName(null);  //use fixture
        assertEquals("setName(null)", -1, returnFromSet);

        String actual = aj.getName();           //use fixture
        assertEquals("setName(null)", "Unit Test", actual);
    }

```

```
    @Test
    public void testSetName_MaxLength10_good() {

        long returnFromSet = aj.setName("123456789-");
        assertEquals("setName(10chars)", 1, returnFromSet);

        String actual = aj.getName();
        assertEquals("setName(10chars)",
            "123456789-", actual);
    }

    @Test
    public void testSetName_ExceedsMaxLength10_bad() {

        long returnFromSet = aj.setName("123456789-1");
        assertEquals("setName(11chars)", -1, returnFromSet);

        String actual = aj.getName();
        assertEquals("setName(11chars)",
            "Unit Test", actual);
    }

    @Test
    public void testSetName_EmptyString_bad() {

        long returnFromSet = aj.setName("");
        assertEquals("setName(0chars)", -1, returnFromSet);

        String actual = aj.getName();
        assertEquals("setName(0chars)",
            "Unit Test", actual);
    }

    @Test
    public void testSetName_MinLength1_good() {

        long returnFromSet = aj.setName("1");
        assertEquals("setName(1char)", 1, returnFromSet);

        String actual = aj.getName();
        assertEquals("setName(1char)",
            "1", actual);
    }

    @Test
    public void testSetName_MiddleLength5_good() {

        long returnFromSet = aj.setName("12345");
        assertEquals("setName(5chars)", 1, returnFromSet);

        String actual = aj.getName();
        assertEquals("setName(5chars)",
            "12345", actual);
    }
}
```


assert METHODS

1. JUnit provides assert methods that compare expected and actual results, for use after each call to a business method.

2. These methods return void, and throw AssertionFailedError if the assertion is false, which is caught by JUnit for reporting in a stack trace with your optional String, and the expected and actual values:

   ```
   java.lang.AssertionError: setName(null) expected:<-1>
   but was:<1>
   ```

3. Assert methods are overloaded in two ways:

   a. An optional first parameter String enables you to code a message to be stored in the AssertionFailedError.

   b. Data types long, double, and Object are supported for comparing expected and actual values for each test.

4. For == with long values, or this.equals(ref) with Objects:
   ```
   assertEquals ("Johnson", employee.getName() );
   assertEquals (expected, actual)
   assertEquals (messageString, expected, actual)
   ```

5. For double values, the allowed difference is required:

   ```
   assertEquals (expected, actual, allowedDiff)
   assertEquals (messageString, expected, actual, allowedDiff)
   ```

6. For asserting that two references point to the same Object:

   ```
   assertSame (ObjectRef1, ObjectRef2)
   assertSame (messageString, ObjectRef1, ObjectRef2)
   ```

7.
   ```
   assertTrue (employee.getBonus() > 1000.00 );
   assertTrue (booleanArg)
   assertTrue (messageString, booleanArg)
   assertFalse (booleanArg)
   assertFalse (messageString, booleanArg)
   ```

8.
   ```
   assertNull (objectRef)
   assertNull (messageString, objectRef)
   assertNotNull (objectRef)
   assertNotNull (messageString, objectRef)
   ```

9. To throw an AssertionFailedError, to stop execution of a test method if the test should have thrown an exception but didn't. Testing continues if there are more test methods.

   ```
   fail ()
   fail (messageString)
   ```

JUnit 4 static import, ANNOTATIONS

static import

1. A static import can be used to import a class and also to import methods.

    a. The static import on line 5 in AJ1708Test is needed because your JUnit test class does not have a reference to a JUnit object, so JUnit methods cannot be called in the usual way via ref.methodname().

ANNOTATIONS

2. Test methods must have the @Test annotation.

3. JUnit annotations can specify when a JUnit method in your test class should be executed:

	Annotation	When is method executed
a.	@Before	Before each test method
b.	@After	After each test method
c.	@BeforeClass	Before any test method or setUp()
d.	@AfterClass	After all test methods are finished

4. Work with test fixtures @BeforeClass and @AfterClass may introduce dependencies between tests, and should be avoided or used with great caution.

OPTIONAL

5. Less frequently used JUnit annotations can specify how a JUnit method in your test class should be executed:

a.	@Ignore	Ignore this test method
b.	@Test(timeout=2000)	Fail this test method if it runs longer than 2000 milliseconds.
c.	@Rule	Alter how a test method is run and/or reported. See the javadoc for package org.junit.rules
d.	@Parameters	Mix of test methods and test data
e.	@Theory	Intended behavior for a large number of scenarios
f.	@Category	Run only selected tests based on a scheme of test method categories

6. The package org.junit contains the class Assert and the annotations After, AfterClass, Before, BeforeClass, and Test.

HAMCREST MATCHERS

1. The name "Hamcrest" is an anagram (rearranged sequence of letters) created from the word "matchers." Hamcrest is a framework (group of Java classes and interfaces) for specifying JUnit-like tests for Java and other languages.

2. Advantages of Hamcrest are ease of specifying complex tests, greater readability, more useful error messages, and the power to combine and negate tests.

3. Comparison of JUnit and Hamcrest:

   a. JUnit: `assertEqual (123, varname);`
   b. Hamcrest: `assertThat (varname, is(123) );`

4. The method assertThat was created by Joe Walnes before May, 2005. His syntax was assertThat (value, matcherStatement);

   a. `assertThat (varname, is(not(1.5)));`
   b. `assertThat (ref.getName(), is("Amy"));`
   c. `assertThat (stringRef, either(containsString("a")).`
      `or(containsString("b")));`
   d. `assertThat (stringRef,`
      `anyOf(containsString("a"), containsString("b")) );`
   e. `assertThat (myArrayList, hasItem("Bob"));`

5. The class org.hamcrest.CoreMatchers contains many methods that provide matcher tests, such as: allOf, any, anyOf, anything, both, containsString, describedAs, either, endsWith, equalTo, everyItem, hasItem, hasItems, instanceOf, is, isA, not, notNullValue, nullValue, sameInstance, startsWith.

6. The class org.hamcrest.core.CombinableMatcher contains many methods that provide matcher tests, such as: and, both, describeTo, either, matchesSafely, or.

7. The assertThat method is the most commonly used.

8. You can write your own custom Matcher by extending the abstract class org.hamcrest.BaseMatcher.

9. See http://junit.org/junit4/javadoc/latest/ The important packages are org.hamcrest and org.hamcrest.core.

10. Examples of "acceptable range" non-specific values (other than random-number-generators) include: freezer compartment temperature, flight simulator time to reach destination when winds vary in speed, truck fuel consumption in terrain with varying inclines, and boat fuel consumption or time to reach destination in unpredictable waters.

HAMCREST EXAMPLE

AJ1713.java

```
package com.themisinc.u17;
public class AJ1713 {
    private String name;

    public AJ1713 (String name) {
        setName (name);
    }

    public void setName (String name) {
        if (name==null) return;
        if (name.length() > 0 && name.length() < 11) {
            this.name = name;
        }
    }
    public String getName () {
        return name;
    }
}
```

AJ1713Test.java

```
package com.themisinc.u17Test;
import com.themisinc.u17.AJ1713;

import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;            //static import

import static org.hamcrest.CoreMatchers.*;    //static import

public class AJ1713Test {

    private AJ1713 holdName;                  //test fixture

    @Before
    public void setUp() throws Exception {
         holdName = new AJ1713 ("Mary");      //initialize
    }                                         //before each
                                              //test method
    @Test
    public void testSetName_Over10Char_bad() {

        holdName.setName("Sarah Malka");

        //NOTE, wrong expected is specified in tests
        //JUnit stops at the first failure per test method

        //assertEquals("Marie", holdName.getName() );

        assertThat(holdName.getName(), is("Marie") );
    }
}
```


2 COMMON WAYS TO ORGANIZE SOURCE FOLDERS FOR TEST CLASSES

1. You can place your test classes under your project src folder in a separate folder tree parallel to your business classes:

```
ProjectHeadDir
    src
        com.themisinc.application
            BusinessClass1
            BusinessClass2
        com.themisinc.test
            BusinessClass1Test
            BusinessClass2Test
    JRE System Library [JavaSE-1.7]
    Referenced Libraries
        junit-4.10.jar - C:\MyJUnit10
```

   a. To do this, when you create a new class, enter one package name for your business classes, such as com.themisinc.application as shown above, and a different package name for your test classes, such as com.themisinc.test as shown above.

   b. Your business classes would need the package statement:

```
package com.themisinc.application;
```

   c. Each of your test classes would need a package statement for itself, and an import statement for the business class that it tests:

```
package com.themisinc.test;
import com.themisinc.application.BusinessClass1;
```

2. You can place your test classes under a test folder that is at the same level as the business class src folder:

```
ProjectHeadDir
    src
        com.themisinc
            BusinessClass1
            BusinessClass2
    test
        com.themisinc
            BusinessClass1Test
            BusinessClass2Test
    JRE System Library [JavaSE-1.7]
    Referenced Libraries
        junit-4.10.jar - C:\MyJUnit10
```

a. The src folder contains business classes to be tested. The business classes would need the package statement:

```
package com.themisinc;
```

b. The test folder contains one test class per business class. The test classes would need the package statement:

```
package com.themisinc;
```

c. <u>To make the folder called test</u>:

1) In Project Explorer, highlight the name of your project, then right-click on it.

2) In the popup, click New, Source Folder.

3) In the "New Source Folder" popup, for "Folder name:" enter the folder name <u>test</u>. Click Finish.

OPTIONAL

3. JUnit has its own jar file with JUnit bytecode. In both folder structures above, the name of the jar file is junit-4.10.jar. The pathname after the jar filename is displayed in gray in Package Explorer; it is the folder where the jar file is located.

4. When you make a folder called test by using the procedure in 2.c. above, Eclipse will treat the test folder as a container for classes and packages.

a. Eclipse adds the test folder to a classpath.

1) Eclipse uses multiple classpath variables. A classpath is an environment variable that lists one or more folders or jar files to be searched to locate classes and packages.

2) Because Eclipse puts src, bin, and test in a classpath, you can have packages with the same name under all three folders.

3) If you work on a commandline instead of Eclipse, to achieve the same result that Eclipse achieves you would enter commandline options when you compile and execute your tests.

b. Compiled bytecode for classes in your source folders src and test will be placed under the bin folder.

ADD THE JUnit JAR FILE TO THE BUILD PATH FOR YOUR PROJECT

1. Eclipse's Java builder builds Java programs using its own compiler (the Eclipse Compiler for Java) that can build programs incrementally as individual source files are saved.

2. The build path for a project is the classpath (list of folders and jar files) to be searched for the purpose of compiling.

3. The JUnit jar file must be added to the build path for a project when you create the first test class for the project. This only has to be done once per project.

4. <u>To use the JUnit 4 that is downloaded with your Eclipse</u>:

   a. If you import your JUnit source code from a jar file, to set the build path you should create a JUnit test class via the procedure on aj17.17 and set the build path. You can delete the new JUnit test class if you don't need it.

   b. If you use Eclipse to create a JUnit test class and manually enter your source code, follow the procedure on aj17.17.

OPTIONAL

5. <u>To add a different version of JUnit to the Eclipse build path for a project</u>:

   a. In Package Explorer, highlight the project name.
   b. Click Project, Properties, Java Build Path.
   c. Click the tab for Libraries.
   d. Click "Add External JARs".
   e. Navigate to the folder that contains your JUnit jar file, highlight the file, and click Open, OK.

6. To see your Eclipse's version of JUnit, in Package Explorer, expand the icon for JUnit 4. The JUnit framework is under the package org.junit, and has two files, for example:

   ```
   C:\myEclipse\eclipse-jee-kepler-R-win32\eclipse\plugins\
   org.junit_4.11.0.v201303080030\junit.jar

   C:\myEclipse\eclipse-jee-kepler-R-win32\eclipse\plugins\
   org.hamcrest.core_1.3.0.v201303031735.jar
   ```

CREATE AND EXECUTE A JUnit 4 TEST

1. To set up a new JUnit 4 test class, assuming:

```
project name:     Java2
business folder:  Java2/src/com/themisinc/u17
business package: com.themisinc.u17
business class:   AJ1708.java
test folder:      Java2/src/com/themisinc/u17Test
test package:     com.themisinc.u17Test
test class:       AJ1708Test.java
test must import: com.themisinc.u17.AJ1708;
```

   a. In Package Explorer, highlight the business class to be tested.
   b. Click File, New, JUnit Test Case.
   c. Keep the selection for "New JUnit 4 test".
   d. For "Source folder:" enter Java2/src
   e. For "Package:" enter com.themisinc.u17Test
   f. "Name:" will be set to AJ1708Test
   g. "Superclass:" will be set to java.lang.Object
   h. For "Which method stubs would you like to create", click the checkbox for setUp().
   i. For "Do you want to add comments?", chose what you like.
   j. "Class under test:" will be set to com.themisinc.u17.AJ1708
   k. Click Next.
   l. In the "Test Methods" window:
      1) Click the method to be tested, setName(String)
      2) Check "Create tasks for generated test methods".
      3) Click Finish.
   m. <u>If you are creating the first JUnit test class for a project, you will get a popup that says: "JUnit 4 is not on the build path. Do you want to add it?"</u>
      1) Click OK to accept the default settings in the popup, which are "Perform the following action:" and "Add JUnit 4 library to the build path".

2. A new JUnit 4 test class will be created and displayed in the Editor. It contains a method stub for setUp() and one test method stub for each method you checked for testing.

   a. Manually change the name of each test method to describe the specific test to be done in that method.
   b. To create more test methods, use control-c to copy and control-v to paste.

3. To execute the test class, click Run, Run As, JUnit Test.

   a. The JUnit Overview view will overlay the Package Explorer and list the tests that passed and failed.
   b. The Failure Trace view provides details about failures. Double-click on a trace line to go to its test case line.
   c. After correcting code errors, test again.

OPTIONAL: OLDER STYLE TEST METHOD, JUnit 4

<u>**AJ1708TestOldStyle.java**</u>

```
package com.themisinc.u17;
import static org.junit.Assert.*;
import org.junit.Test;
import org.junit.Before;
public class AJ1708TestOldStyle {

    private AJ1708 aj;
    @Before
    public void setUp() throws Exception {
        aj = new AJ1708 ("Unit Test");
    }

    @Test
    public void testSetName() {       //test setName, getName
        long returnValue;
        String actual;

        returnValue = aj.setName(null);           //null//bad
        assertEquals("setName(null)", -1, returnValue);
        actual = aj.getName();
        assertEquals("setName(null)",
            "Unit Test", actual);       //depends on setUp()

        returnValue = aj.setName("12345");    //5 chars//good
        assertEquals("setName(5chars)", 1, returnValue);
        actual = aj.getName();                //no dependency
        assertEquals("setName(5chars)", "12345", actual);

        returnValue = aj.setName("123456789-A");  //> 10//bad
        assertEquals("setName(11chars)", -1, returnValue);
        actual = aj.getName();
        assertEquals("setName(11chars)",
            "123456789-", actual); //depends on previous test
    }
}
```

==

1. An older JUnit 4 style was to put all tests for one business method into one test method, to localize the tests and control the sequence in which they executed. But this style created <u>dependencies</u> where the outcome of a test depended on the outcome of a previous test. This style was error prone because you had to follow the changes in the tested value from test to test.

   a. If any assert method fails, the current test method stops executing with AssertionFailedError. Execution continues with other test methods that have not yet been executed.

TEST FIXTURES AND TEARDOWN

1. A test fixture is a set of one or more variables, Objects, files, and/or databases containing a known, fixed set of values, to be used as a baseline for running tests.

   a. Use of test fixtures ensures that the tests are run in a known, fixed environment, so that tests and results are repeatable.

   b. Use of test fixtures enables you to:

      1) Separate testing from test initialization.

      2) Reuse a known state (set of values) for more than one test.

      3) Create a setUp procedure once and reuse it.

2. If your business methods to be tested are instance methods, the test class must create an object of the business class type, with a reference pointing to it. The object and reference would be test fixtures.

3. Test fixtures can be created in each test method, but JUnit 4's setUp method is the preferred place to create them, especially if they are required by multiple test methods.

   a. The setUp method is called before each test method runs. Then the test method executes and uses the fixture(s).

<u>tearDown METHOD</u>

4. JUnit 4 has a tearDown method that is called after each test method executes, which can clean up after each test method.

   a. The tearDown method is not needed for reference test fixtures because the setUp method assigns each reference to point to a new object, and the old object is garbage collected normally.

   b. The tearDown method is useful to clean up database rows that were added or modified by a test method or by the business method it called.

5. In JUnit 3, both setUp and tearDown methods threw java.lang.Exception. Many people still code "throws Exception" in the method headers for these methods in JUnit 4.

TESTS FOR CONSTRUCTORS

1. If a constructor only calls set methods, do not test the set methods again as a test of the constructor.

    a. The test method for this kind of constructor should create an object and then test the values in the variables that the constructor affects.

    b. Constructor:

    ```
    public Product (long number, String name) {
        setNumber (number);
        setName (name);
    }
    ```

    c. Test method:

    ```
    @Test
    public void testProduct_LongString () {
        Product p = new Product (243L, "5' cable");
        assertEquals (243L,       p.getNumber() );
        assertEquals ("5' cable", p.getName() );
    }
    ```

2. When the business class has overloaded constructors, a good style is to include the parameter types in the name of the test method, as shown above.

TESTS WITH EXCEPTION HANDLING

<u>business methods</u>

```
public void setSeats (int seats) throws BadDataException {
    if (seats < 1) {
        throw new BadDataException ("seats < 1");
    }
    this.seats = seats;
}
public int getSeats () {
    return seats;
}
```

<u>test method</u>

```
1   public void testSetSeats() throws BadDataException {
2       try {
3           aj.setSeats (789);
4       } catch (BadDataException e) {
5           //e.printStackTrace(); //may help to find the error
6           fail ("valid seats 789 shd not cause Exception");
7       }
8       assertEquals ("valid seats 789", 789, aj.getSeats() );
9
10      try {
11          aj.setSeats (0);
12          fail ("invalid seats 0 shd cause Exception");
13      } catch (BadDataException e) {
14          assertEquals ("seats < 1", e.getMessage() );
15      }
16  }
```

---

1. The test method's header has a throws clause, but if the exception is thrown other than in a try-catch, execution of the test method stops, and JUnit continues with other test methods that have not yet been executed. In the test method:

    a. Line 6: fail because the business method should not throw an exception when it receives good data.
    b. Line 12, fail because the business method should have thrown an exception when it received bad data.
    c. Line 14: ensure that the correct message is included in the exception that JUnit throws for the business method.

2. The code below is the wrong way to use JUnit, and causes JUnit's red line and failure trace due to BadDataException.

```
a   public void testSetSeats() throws BadDataException {
b       aj.setSeats(0);
c       assertEquals("setSeats(0)", 8, aj.getSeats());
d   }
```


TEST SUITES, JUnit 4

1. Running many test classes for an application is easier when they are grouped together into a test suite.

    a. When the test suite is executed, each test class in the test suite is executed.

    b. Test suites can contain other test suites.

2. <u>Create a JUnit 4 test suite</u>:

    a. In Package Explorer, highlight the source folder of the test classes. Click File, New, Other....

    b. For "Wizards:" enter JUnit. Select "JUnit Test Suite". Click Next>. In "Test classes to include in suite:" select the names you want. Click Finish.

    c. Alternatively, you can manually create a regular class (not a test class) under your test source folder and manually enter the syxtax shown below.

3. To execute the test suite click Run, Run As, JUnit Test.

<u>AllTests4.java, Test Suite in JUnit 4</u>

```java
package com.themisinc.u17Test;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses ( {                   //open paren, open curly
    AJ1708Test.class,               //comma-separated list
    AJ1723Test.class,
} )                                 //close curly, close paren

public class AllTests4 {            //empty class
}
```

BUSINESS AND TEST CLASSES FOR JUnit 4 TEST SUITE

AJ1723.java

```java
package com.themisinc.u17;
public class AJ1723 {

    private int seats;

    public AJ1723 (int s) {
        setSeats (s);
    }
    public boolean setSeats (int s) {
        if (s > 0 && s < 15) {          //valid range is 1-14
            this.seats = s;
            return true;
        }
        return false;
    }
    public int getSeats () {
        return seats;
    }
}
```

AJ1723Test.java

```java
package com.themisinc.u17Test;
import com.themisinc.u17.AJ1723;

import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;

public class AJ1723Test {

    private AJ1723 aj;

    @Before
    public void setUp() {
        aj = new AJ1723 (8);
    }

    @Test
    public void testSetSeats_Zero_bad() {

        assertEquals("setSeats(0)", false, aj.setSeats(0));

        assertEquals("setSeats(0)", 8,     aj.getSeats());
    }
}
```


OPTIONAL: JUnit 3 IS STILL SUPPORTED

1. To create a JUnit3 test class and use the version of JUnit 3 that is downloaded with your Eclipse:

    a. In Package Explorer, highlight the folder where you want the test class. Click File, New, JUnit Test Case.

    b. Select the button for "New JUnit 3 test".

    c. Fill in the entry areas the same as for JUnit 4, except:

        1) For "Superclass:" enter junit.framework.TestCase

        2) For your first JUnit test class for a project, you will be prompted by a popup that says: "JUnit 3 is not on the build path. Do you want to add it?". The default settings in the popup are "Perform the following action:" and "Add JUnit 3 library to the build path". Click OK.

2. For JUnit 3.8 and earlier, the JUnit jar is under the junit.framework package.

3. The TestCase class is documented at http://junit.sourceforge.net/junit3.8.1/javadoc/junit/framework/TestCase.html

4. Your test class header must explicitly extend TestCase.

5. Your test methods MUST have names starting with "test"

    a. JUnit 3 does not use annotations.

    b. Test methods execute in unpredictable sequence, not necessarily in the order they are coded.

    c. JUnit uses Reflection to locate the methods to be executed for the test.

6. The setup method is automatically called prior to each test method.

7. To execute a test, click Run, Run As, JUnit Test

OPTIONAL: JUnit 3 EXAMPLE

<u>AJ1725.java</u>

```
package com.themisinc.u17;
public class AJ1725 {

    private int seats;

    public AJ1725 (int s) {
        setSeats (s);
    }
    public boolean setSeats (int s) {
        if (s > 0 && s < 15) {          //valid range is 1-14
            this.seats = s;
            return true;
        }
        return false;
    }
    public int getSeats () {
        return seats;
    }
}
```

<u>AJ1725Test3.java</u>

```
package com.themisinc.u17;
import junit.framework.TestCase;
public class AJ1725Test3 extends TestCase {

    private AJ1725 aj;

    protected void setUp() throws Exception {
        super.setUp(); //optional line created by Eclipse
        aj = new AJ1725 (8);
    }

    public void testSetSeats_Zero_bad() {

     int     param1            = 0;
     String  setFailString     = "setSeats(0)";
     boolean expectedSetReturn = false;

     String  getFailString     = "getSeats() after set to 0";
     int     expectedGetReturn = 8;

        assertEquals(setFailString,
            expectedSetReturn, aj.setSeats(param1) );

        assertEquals(getFailString,
            expectedGetReturn, aj.getSeats() );
    }
}
```


OPTIONAL: JUnit 3 TEST SUITES

1. Running many test classes for an application is easier when they are grouped together into a test suite.

    a. When the test suite is executed, each test class in the test suite is executed.

    b. Test suites can contain other test suites.

2. Create a JUnit 3 test suite: in Package Explorer, highlight the source folder of the test classes. Click File, New, Other.... For "Wizards:" enter JUnit. Select "JUnit Test Suite". Click Next>. In "Test classes to include in suite:" select the names you want; classes defined with "extends TestCase" are pre-selected. Click Finish.

    a. AllTests is the default name for the new test suite. The name can be changed.

    b. After the test suite is created, test classes can be added or dropped manually. Alternatively you can use the Package Explorer and right click on the test suite, then click "Recreate Test Suite".

    c. To control the order of calling test methods, manually enter lines like lines 15 and 16 below.

3. To execute the test suite click Run, Run As, JUnit Test.

AllTests3.java, Test Suite in JUnit 3

```
1   package com.themisinc.u17;
2
3   import junit.framework.Test;
4   import junit.framework.TestSuite;
5
6   public class AllTests3 {
7
8       public static Test suite() {
9           TestSuite suite = new TestSuite (
10              AllTests3.class.getName() );
11          //$JUnit-BEGIN$
12          suite.addTestSuite (CustomerTest.class);
13          suite.addTestSuite (ProductTest.class);
14          suite.addTestSuite (OrderTest.class);
15          suite.addTestSuite (new ProductTest ("testSetName");
16          suite.addTestSuite (new OrderTest   ("testSetShelf");
17          //$JUnit-END$
18          return suite;
19      }
20  }
```

EXERCISES

Notes
No solutions are provided for this exercise.
See 17.16 and 17.17 for the procedures to set the build path and to create and execute a JUnit 4 test.

1. A copy of business class AJ1708.java should be in your com.themisinc.u17. A copy of the test class AJ1708Test.java should be in your com.themisinc.u17Test.

    a. Execute the test.

    b. Make the following error in the code, and use the test class to find it.

    ```
    if name.length() < 0
    ```

2. Create and run a test suite.

    a. A copy of business class AJ1723.java should be in your com.themisinc.u17. A copy of the test class AJ1723Test.java should be in your com.themisinc.u17Test.

    b. Create a test suite for both AJ1708Test.java and AJ1721Test.java

    c. Run the test suite.

OPTIONAL: @Category AND @IncludeCategory

1. JUnit 4 introduced two annotations to control which classes and methods are tested by a test suite, and to add metadata on the tests.

   a. First, the annotation @Category is applied to methods or classes in the test suite. This annotation alone has no effect. Above, the three categories are Fast, Medium, and Slow. Either classes or interfaces can be used as categories.

   b. Second, the annotation @IncludeCategory specifies to run only the classes and methods annotated with a given category or a subtype of that category.

2. To exclude categories, use the @ExcludeCategory annotation

3. The facing page shows how to run only one category. Below is an example of running multiple categories. Tests that would be run are ATest's cMethod, and BTest's dMethod.

```
a   @RunWith(Categories.class)
b   @IncludeCategory({
c       Fast.class,
d       Medium.class
e   })
f   @SuiteClasses({
g       ATest.class,
h       BTest.class
i   })
j   public static class FastAndMediumTestSuite {
k   }
```

4. Commonly used categories are:

   a. Type of test: Unit, Integration, Smoke (preliminary test to determine whether to test more deeply), Regression, Performance, etc.
   b. Speed of execution: Slow, Quick, etc.
   c. When tests should be executed: NightlyBuildTests, etc.
   d. State of the test: Unstable, InProgress, etc.
   e. Project specific metadata to specify what feature of a project is covered by the test.

OPTIONAL: @Category AND @IncludeCategory, EXAMPLE

```
public interface Fast {            //category marker interface
}                                  //with no methods or constants
public interface Medium
}
public interface Slow {
}

public static class ATest {
    @Test
    public void aMethod () {                        //will NOT run
        fail();
    }
    @Category(Slow.class)                               //will run
    @Test
    public void bMethod () {
    }
    @Category({Fast.class, Medium.class})      //Will NOT run
    @Test
    public void cMethod () {
    }
}

@Category({Fast.class,Slow.class})   //Applies to whole BTest
public static class BTest {
    @Test
    public void dMethod() {
    }
}

@RunWith(Categories.class)    //Categories is a kind of Suite
@IncludeCategory(Slow.class)         //only run Slow category
@SuiteClasses({
    ATest.class,
    BTest.class
})
public static class SlowTestSuite {       //one category only
}
```


OPTIONAL: @Test (expected=ExceptionName.class) AND @Rule

@Test (expected=ExceptionName.class)

1. The @Test annotation has an optional parameter expected that accepts values that are subclasses of Throwable.

```
@Test(expected = IndexOutOfBoundsException.class)
public void errorWithArrayList () {
    new ArrayList<Object>().get(0);  //no element in new list
}
```

2. Limitations of the expected parameter of @Test:

   a. It passes if any code in the method throws the specified exception, so it may be useful only with the shortest, simplest tests.

   b. You can't test the value of the message in the exception, or values in the test fixture bject after the exception has been thrown.

@Rule AND ExpectedException

3. You can specify what exception and exception message you expect by using the ExpectedException rule.

4. When the test on the facing page fails, the JUnit message is: java.lang.AssertionError: Expected test to throw (an instance of com.themisinc.u17.NullNameException and exception with message a string containing "null name")

5. Another example:

```
a   @Rule
b   public ExpectedException thrown = ExpectedException.none();
c   @Test
d   public void shouldTestExceptionMessage()
e       throws IndexOutOfBoundsException {
f
g       thrown.expect(IndexOutOfBoundsException.class);
h       thrown.expectMessage("Index: 0, Size: 0");
i
j       List<Object> list = new ArrayList<Object>();
k       list.get(0);        //method execution stops at this line
l       //other statements
m   }
```

6. thrown.expectMessage allows use of Hamcrest Matchers such as:

```
thrown.expectMessage(Matchers.containsString("Size: 0"));
```

OPTIONAL: @Test AND @Rule, EXAMPLE

AJ1798a.java

```
package com.themisinc.u17;
public class AJ1708a {
    private String name;
    public AJ1708a (String name) throws NullNameException {
        setName (name);
    }
    public int setName(String name)throws NullNameException {
        //if (name==null) throw new NullNameException
            ("null name");
        if (name==null)  return -1;
        if (name.length() > 0 && name.length() < 11) {
            this.name = name;
            return 1;
        }
        return -1;
    }
    public String getName () {
        return name;
    }
}
```

AJ1708ExceptTest.java

```
package com.themisinc.u17Test;
import com.themisinc.u17.AJ1708a;
import com.themisinc.u17.NullNameException;
import org.junit.Before;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.ExpectedException;
import static org.junit.Assert.*;
public class AJ1708ExceptTest {
    private AJ1708a aj;                                 //fixture
    @Before
    public void setUp() throws Exception {              //SetUp()
         aj = new AJ1708a ("Unit Test");
    }
    @Rule
    public ExpectedException thrown=ExpectedException.none();
    @Test
    public void testSetName_NullValue_bad ()
    throws NullNameException {

        long returnFromSet = aj.setName(null);
        thrown.expect(NullNameException.class);//these can be
        thrown.expectMessage("null name");     //anywhere in
                                               //the methhod
        assertEquals("setName(null)", -1, returnFromSet);
        String actual = aj.getName();
        assertEquals("setName(null)", "Unit Test", actual);
    }
```


(blank)

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. Create documentation comments in a program, use the javadoc documentation generator to generate documentation for the program, and display the documentation in a browser.

THE jar UTILITY

1. The name "jar" stands for "java archive." A jar archive contains one or more other files.

    a. Jar files do not have to be compressed, but typically they are. Jar uses the same compression algorithm as Winzip and other Windows zip compression software.

    b. Jar and zip files are internally the same, so either the jar utility program or a zip program can be used with them. You may change the filename extension from .jar to .zip, or vice versa, depending on which software you use.

    c. Eclipse procedures for jar files are on pages E.16-E.17.

2. Jar files are often attached to an email, or placed on a shared drive, to help developers work with the same code.

3. 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.

4. The Java API source code is stored in a directory tree in a jar archive called either src.jar or src.zip.

5. Java classes to read and write jar archives are in the package java.util.zip.

6. 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)
```

7. jar options do not use - hyphen. 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
```

jar ON A COMMANDLINE

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 embedded within source code files eliminates the problem of separate internal and external documentation.

2. Documentation comments can apply to classes, methods, and member variables. Constructors are documented like methods.

   a. A documentation comment must immediately precede the class or method or member variable that it applies to.

3. javadoc.exe is a utility program that is part of the JDK download. Javadoc uses documentation comments in source files to build a linked set of HTML files, one per class, and an index and hierarchy tree, to be viewed in a browser.

   a. javadoc works ONLY on the source files that you specify when you run javadoc.

   b. By default only public classes, and public and protected members, are documented unless you request private members also. On a command line this is done via the -private flag.

   c. You can enter a Style sheet (xxx.css file).

4. For online tutorials do a search on oracle tutorial javadoc

5. Execute the javadoc documentation generator:

   a. `$ javadoc  Class1.java  Class2.java          ---UNIX`
   b. `C:\myjava> javadoc  Class1.java  Class2.java  ---DOS`
   c. Eclipse procedures for javadoc are on page E.23.

6. Two ways to view the documentation in a browser:

   a. In Windows Explorer, open your javadoc 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:\tah\CaseStudy\doc\index.html

   c. Internet Explorer is preferred. Some embedded javadoc tags may not work in Firefox or Safari.

   d. After the documentation is displayed in your browser, you may have to click Frames. Then click on your desired package or class.

AJ1805.java

```java
package com.themisinc.u18;

/** The <code>AJ1805</code> class contains javadoc comments.
*   @author   Teresa Alice Hommel
*   @since    3/31/11
*/
public class AJ1805 {

    protected static TCourse1805 tcSeats;
    protected static TCourse1807 tcName;

    /** The <code>main</code> method instantiates objects for
    *   the two static references. The data is printed.
    *   @param    args   Commandline arguments in a String[]
    */
    public static void main (String[] args) {
        tcSeats = new TCourse1805 (12);
        tcName  = new TCourse1807 ("Java");
        System.out.println (
          tcName.getName() + " has " + tcSeats.getSeats() );
    }
}
```

TCourse1805.java

```java
package com.themisinc.u18;

/** The <code> TCourse1805 </code> class continues the demo
*   of javadoc comments. Note, only the first sentence goes
*   in a method or field Summary. All sentences go in the
*   method or field Detail.
*   @version  1.0
*/
public class TCourse1805 {

    /** seats for training course. It is protected.*/
    protected int seats;

    /** Constructor initializes the int. A value must
    *   be passed because there is only one constructor.
    *   @param    seats  int number of students in course.
    */
    public TCourse1805 (int seats) {
        this.seats = seats;
    }

    /** Typical get method. The int seats is returned.
    *   @param    none
    *   @return   int
    */
    public int getSeats() {
        return seats;
    }
}
```

javadoc AND HTML

1. Documentation comments may contain HTML tags. HTML header tags such as <H1> <H2> etc. should not be used because they can interfere with javadoc's generated headers.

2. Some HTML tags commonly found in documentation comments are:

    a. <code> text </code>
    The text will display in a font suitable for programming code, typically monospaced font ("Currier").

    b. text
    The text will display in bold. The tags and are recommended, but   are more popular.

    c. <i> text </i>
    The text will display in italic. The tags and (emphasis) are recommended, but <i> </i> are more popular.

    d. <u> text </u>
    The text will display as underlined.

    e. <p>
    Causes a line break and a blank line to be created for the end of a paragraph.

    f.
    Causes a line break. Subsequent text is on next line.

    g. <hr>
    Causes a horizontal line across the page.

    h. <blockquote> text </blockquote>
    The text will display as a separate indented paragraph.

    i. <pre> text </pre>
    Use these tags for text that is preformated, such as programming code. The lines will display as is, with indentation and line breaks, rather than being concatenated and then wrapped into paragraph form.

    j. text
    The text will appear as a link to the specified url. You must manually try the hyperlink to see if it works.

    k.
```
<ol>                              <ul>
    <li> first item                   <li> first item
    <li> second item                  <li> second item
</ol>                             </ul>
```
    Ordered lists are numbered. Unordered lists have bullets. Lists contain list items that use only the tag <li>.

TCourse1807.java

```
package com.themisinc.u18;

/** The <code>TCourse1807</code> class has <i>javadoc</i>
*   comments. This class holds the <u>name</u> of the
*   <b>training course</b>.<br>
*   Valid names are
*   <hr>
*   <ol>
*       <li> Java
*       <li> UNIX
*       <li> HTML
*   </ol>
*   <hr>
*
*   To learn more about java, please
*   <a href="http://docs.oracle.com/javase/7/docs/api/">
*   click here</a>.
*
*   @author   Teresa Alice Hommel
*/
public class TCourse1807 {

    /** By default, documentation comments for private
    *   members are not included in javadoc documentation.
    *   You can request their inclusion.
    */
    private String name;

    /** Reservations are confirmed upon receipt of a correct
    *   coursename. Create your coursename String as follows:
    *   <pre>
    *       String s;
    *       s = reserveCourse(int yourEmpNo);
    *   </pre>
    *   <p>
    *   Please email the ReservationCenter to correct errors.
    *   <p>
    *   <blockquote>
    *   Contact:
    *   trainingCenter@training.com
    *   </blockquote>
    */
    public TCourse1807 (String name) {
        this.name = name;
    }

    /** Call getName() to obtain the course name.
    *   @return   int
    */
    public String getName () {
        return name;
    }
}
```


javadoc TAGS, @ AND LOWERCASE LETTERS

javadoc tag	used for	purpose, notes
@author text	classes	Author of classs, his/her email, etc. Multiple @author tags must be consecutive. May need to use javadoc -author
@deprecated text	classes methods variables	Deprecated items should not be used in new code. The tag causes the compiler to issue a warning if the item is used. This javadoc tag is superseded by the @Deprecated annotation introduced in Java 5.
@param name text	methods	Name of a parameter to a method. One @param should be coded for each parameter to be received. The @param tags should be in the same order as the parameter list.
@return text	methods	Value returned by a method
@see classLink @see methodLink	classes methods variables	Hyperlink to other documentation. For a class, the link can be relative or fully-qualified. For a method, the link must be in the form ClassName#methodName
@since text	classes methods variables	Release date when item was created
@throws AException	methods	One @throws should be coded for each exception a method can throw
@version text	classes	Version of a class, or any info. May need to use javadoc -version

PACKAGE INFORMATION

<u>package.html</u>

```
<!DOCTYPE html>
<HTML>

<HEAD>
<TITLE>
package javadoc
</TITLE>
</HEAD>

<BODY>
This package contains classes
that perform the following functions.
Put your list of files and information here.
</BODY>

</HTML>
```

1. The example above shows the contents of a package.html file.

2. Your javadoc documentation can include information about a package, to describe its overall purpose and the capabilities of the classes in it.

3. To do this, create a file called package.html in the package. In Eclipse, create the file by clicking File, New, File.

4. When you view the javadoc in a browser, click on <u>Package</u> to display your package information.

EXERCISES

No solutions are provided for these exercises.

Eclipse procedures for javadoc are on page E.23.

1. Execute javadoc one time to generate HTML documentation for the classes on pages aj18.05 and aj18.07. Then view the result in a browser.

2. OPTIONAL Add javadoc comments to the four classes used in the Unit 6 Case Study exercise. Each comment should have two or more sentences. A sentence is one or more words followed by . period. Then execute javadoc one time to generate HTML documentation for the four classes, and view the result in a browser.

UNIT 19: JAVABEANS PART 2, SORT AND COMPARE COLLECTIONS

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

1. Create JavaBean classes that implement the Cloneable, Comparator, and Comparable interfaces, contain the method clone for a deep or shallow clone, and contain a copy constructor for a deep or shallow copy.

2. Briefly explain the difference between a deep and shallow clone or copy.

CLONES, SHALLOW VERSUS DEEP COPY

1. A clone is a copy of an object made by a clone method, rather than by a copy constructor.

2. A clone can consist of a shallow or deep copy.

   a. A shallow copy of an object contains a copy of every variable in the original, including a copy of each reference, so the original and copied references both point to the same object.

   b. In a deep copy, each copied reference points to a new object that contains a copy of the original pointed-to object. Thus, references in a deep copy are not copies of the original references.

   c. Variables of basic types, and immutable objects such as Strings, do not need deep copying.

SHALLOW COPY

```
 ____________________          ____________________
|                    |        |                    |
|     original       |        |     copied         |
|        ref         |        |     ref            |
|____________\_______|        |____/_______________|
              \                   /
               \_________________/
                |                  |
                |    int i = 15    |
                |__________________|
```

DEEP COPY

```
 ____________________          ____________________
|                    |        |                    |
|     original       |        |        new         |
|       ref          |        |        ref         |
|________|___________|        |________|___________|
         |                             |
  _______|_________             _______|_________
 |                 |           |  copied object  |
 |   int i = 15    |           |    int i = 15   |
 |_________________|           |_________________|
```


Cloneable INTERFACE AND THE Clone METHOD OF Object

1. JavaBeans should override the clone method of Object if clones will need to be created.

2. The clone method of Object returns a shallow copy of an object, which is produced by making a bitwise copy of all instance data.

3. The clone method in Object is protected, so a method that overrides it must be public or protected.

4. By convention, to obtain a clone, a class and each of its superclasses should call super.clone().

```
Policy4 p = (Policy4) super.clone();
```

5. The class of the object to be cloned must implement the Cloneable interface. This is a marker interface (it has no methods) that signifies permission to make a clone.

   a. If Cloneable is not implemented, the cloneNotSupportedException is thrown.

   b. The Object class does not implement Cloneable, so calling the clone method on an object of type Object results in an exception at run time.

6. Immutable objects, such as String objects, do not require a deep copy because when you change the data in a String object you always get a new object, and the old one is garbage collected.

COPY CONSTRUCTORS

7. As an alternative to a clone method, a copy constructor can make a copy of an object. A copy constructor accepts a parameter that is a reference to an object of its own type, copies each instance variable, and and can create either a shallow or deep copy. See also aj4.13.

clone METHOD, DEEP COPY EXAMPLE

AJ1904.java

```
public class AJ1904 {
    public static void main (String[] args) {

        Policy4 p1 = new Policy4 ("WL", 1, 60);
        Policy4 p2 = null;
        try {
            p2 = (Policy4) p1.clone();
            p2.setCVPaymentMode(4);  //modify the deep copy
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        System.out.println  ("p1: " + p1 + "\np2: " + p2);
    }
}
```

Policy4.java

```
1   import java.io.Serializable;
2   public class Policy4 implements Serializable, Cloneable {
3
4       private String policyNo;
5       private ContractVars cv;
6
7       public Policy4(String p,int paymentMode,int graceDays) {
8           this.policyNo = p;
9           setCV (paymentMode, graceDays);
10      }
11      public void setCV (int paymentMode, int graceDays) {
12          cv = new ContractVars (paymentMode, graceDays);
13      }
14      public void setCVPaymentMode (int paymentMode) {
15          cv.setPaymentMode (paymentMode);
16      }
17
18     @Override
19     protected Object clone()throws CloneNotSupportedException{
20          Policy4 p = (Policy4) super.clone();
21          p.setCV (cv.getPaymentMode(),cv.getGraceDays() );
22          return p;
23      }   //line 21 assigns reference variable cv in the clone
24          //to point to a new ContractVars object that contains
25          //copies of this Policy4's ContractVars' data values.
26
27      @Override
28      public String toString () {
29        return "Policy4:policyNo=" + policyNo + "[" + cv + "]";
30      }
31  }
```

Result, AJ1904.java

```
p1: Policy4:policyNo=WL[ContractVars:paymentMode=1,graceDays=60]
p2: Policy4:policyNo=WL[ContractVars:paymentMode=4,graceDays=60]
```

ContractVars.java

ContractVars.java

```
import java.io.Serializable;
public class ContractVars implements Serializable {

    private int paymentMode;
    private int graceDays;

    public ContractVars (int paymentMode, int graceDays) {
        setPaymentMode (paymentMode);
        setGraceDays (graceDays);
    }

    public void setPaymentMode (int paymentMode) {
        this.paymentMode = paymentMode;
    }
    public int getPaymentMode () {
        return paymentMode;
    }

    public void setGraceDays (int graceDays) {
        this.graceDays = graceDays;
    }
    public int getGraceDays () {
        return graceDays;
    }

    @Override
    public String toString () {
        return "ContractVars:paymentMode=" + paymentMode +
            ",graceDays=" + graceDays ;
    }
}
```


OPTIONAL: COPY CONSTRUCTOR, DEEP COPY EXAMPLE

AJ1906.java

```
public class AJ1906 {
    public static void main (String[] args) {
        Policy6 p1 = new Policy6 ("WL", 1, 60);
        Policy6 p2 = new Policy6 (p1);
        p2.setCVPaymentMode (6);
        System.out.println ("p1: " + p1 + "\np2: " + p2);
    }
}
```

Policy6.java

```
import java.io.Serializable;
public class Policy6 implements Serializable, Cloneable {

    private String policyNo;
    private ContractVars cv;

    public Policy6(String p,int paymentMode,int graceDays) {
        this.policyNo = p;
        setCV (paymentMode, graceDays);
    }
    public Policy6 (Policy6 p) {            //deep copy
        this (p.getPolicyNo(),
            p.getCVPaymentMode(), p.getCVGraceDays() );
    }

    public String getPolicyNo() {
        return policyNo;
    }
    public void setCV (int paymentMode, int graceDays) {
        cv = new ContractVars (paymentMode, graceDays);
    }
    public int getCVPaymentMode () {
        return cv.getPaymentMode();
    }
    public void setCVPaymentMode (int paymentMode) {
        cv.setPaymentMode(paymentMode);
    }
    public int getCVGraceDays () {
        return cv.getGraceDays();
    }
    @Override
    public String toString () {
        return "Policy6:" + policyNo + "[" + cv + "]";
    }
}
```

Result, AJ1906.java

```
p1: Policy6:WL[ContractVars:paymentMode=1,graceDays=60]
p2: Policy6:WL[ContractVars:paymentMode=6,graceDays=60]
```


COPY CONSTRUCTOR, SHALLOW COPY EXAMPLE

AJ1907.java

```
public class AJ1907 {
    public static void main (String[] args) {

        Policy7 p1 = new Policy7 ("WL", 1, 60);
        Policy7 p2 = new Policy7 (p1);
        p2.setCVPaymentMode (2);

        System.out.println ("p1: " + p1 + "\np2: " + p2);
    }
}
```

Policy7.java

```
import java.io.Serializable;
public class Policy7 implements Serializable, Cloneable {

    private String policyNo;
    private ContractVars cv;

    public Policy7(String policyNo, ContractVars cv) {
        setPolicyNo (policyNo);
        setCV (cv);   //copy cv reference for a shallow copy
    }

    public Policy7(String p,int paymentMode,int graceDays) {
        this (p, new ContractVars (paymentMode, graceDays));
    }

    public Policy7(Policy7 p) { //p.getCV() gets the ref for
        this (p.getPolicyNo(), p.getCV() ); //a shallow copy
    }

    public String getPolicyNo() {return policyNo;}
    public void setPolicyNo(String p) {policyNo = p;}

    public ContractVars getCV () {return cv;}
    public void setCV (ContractVars cv) {this.cv = cv;}

    public void setCVPaymentMode (int paymentMode) {
        cv.setPaymentMode (paymentMode);
    }

    @Override
    public String toString () {
        return "Policy7:" + policyNo + "[" + cv + "]";
    }
}
```

Result, AJ1907.java

```
p1: Policy7:WL[ContractVars:paymentMode=2,graceDays=60]
p2: Policy7:WL[ContractVars:paymentMode=2,graceDays=60]
```


SORTING: Comparable AND compareTo, Comparator AND compare

1. Sorting objects of a class that has one or more sort fields requires a method that can compare the sort fields of two objects. For each pair of objects, the method must return:

```
negative int    first object is lower than second
zero int        first object is equal to second
positive int    first object is higher than second
```

2. Java has two interfaces that standardize comparing the sort fields of two objects of the same class.

   a. The class can implement Comparable by having a method called compareTo(). A class's compareTo method is its natural comparison method and the sequence of elements it creates is the class's natural ordering.

   b. The class can have one or more related classes which implement Comparator via a method called compare().

3. Classes that implement Comparable include: all wrapper classes for basic data types, Date, String.

4. Terminology: sorting "imposes an order" on the elements in a Collection, or "creates a total ordering on the objects" so each object has a fixed, determinable place in the sequence.

sort methods of the Collections class

5. The java.util.Collections class contains only public static methods, including methods to sort collections. The class of the objects to be sorted must either implement Comparable or have an associated Comparator class.

   a. For ascending order, when all elements implement Comparable and compareTo():

```
Collections.sort (list);
```

   b. For descending order, when all elements implement Comparable and compareTo(), via the Collections method public static Comparator reverseOrder():

```
Collections.sort (list, Collections.reverseOrder());
```

   c. To sort via a Comparator:

```
MyComparator mc = new MyComparator();
Collections.sort (list, mc);
```

Comparable INTERFACE, Collections.sort, EXAMPLE

AJ1909.java

```
import java.util.ArrayList;
import java.util.Collections;
public class AJ1909 {
    public static void main (String[] args) {

/*1*/   ArrayList<Double> a = new ArrayList<Double> ();
        for (int i=0; i<3; i++)
            a.add (0.5 + i);              //autobox to Double

/*2*/   for (Double elem : a)
            System.out.print (elem + "  ");
        System.out.println ();

/*3*/   Collections.sort (a, Collections.reverseOrder());

/*4*/   for (Double elem : a)
            System.out.print (elem + "  ");
        System.out.println ();

/*5*/   Collections.sort (a);

/*6*/   for (Double elem : a)
            System.out.print (elem + "  ");
        System.out.println ();
    }
}
```

Result, AJ1909.java

```
0.5  1.5  2.5
2.5  1.5  0.5
0.5  1.5  2.5
```

===

1. Lists of objects whose class implements Comparable can use Collections.sort() and Collections.reverseOrder(), as shown above.

2. When comparing objects in a collection, the natural ordering for the class should be consistent with the class's equals method, which means the equals and compareTo methods both find the same objects to be equal.

3. The value null is not a class or object. Comparing any object to null using compareTo causes a NullPointerException.

SORT ARRAYLIST VIA Comparator, EXAMPLE

<u>AJ1910.java</u>

```
import java.util.ArrayList;
import java.util.Collections;
public class AJ1910 {
    public static void main (String[] args) {

        ArrayList<Data10> a = new ArrayList<Data10>();
        a.add (new Data10 (20, 94));
        a.add (new Data10 ( 6, 55));
        a.add (new Data10 (20, 82));
        a.add (new Data10 ( 6, -3));

        Data10Comparator dc = new Data10Comparator();
        Collections.sort (a, dc);

        for (Object o : a)
            System.out.print (o + "  ");
        System.out.println ();
    }
}
```

<u>Data10.java</u>

```
public class Data10 {
    private int i;
    private int j;

    public Data10 () {
    }
    public Data10 (int i, int j) {
       this.i = i;
       this.j = j;
    }
    public int getI () {
        return i;
    }
    public int getJ () {
        return j;
    }
    public String toString () {
        return "Data10:" + i + "," + j;
    }
    public boolean equals (Object o) {
        if (o instanceof Data10
        && this.i == ((Data10)o).getI()
        && this.j == ((Data10)o).getJ())
            return true;
        else
            return false;
    }
}
```


Data10Comparator.java

```
import java.util.Comparator;
public class Data10Comparator implements Comparator<Data10> {

    /**sort ascending i as field 1, ascending j as field 2*/
    public int compare(Data10 d1, Data10 d2)
    throws ClassCastException {
        int returnVal = 0;

        if (d1.getI() < d2.getI() ) returnVal = -1;
        if (d1.getI() > d2.getI() ) returnVal = 1;

        if (d1.getI() == d2.getI() ) {
            if (d1.getJ() < d2.getJ() ) returnVal = -1;
            if (d1.getJ() == d2.getJ() ) returnVal = 0;
            if (d1.getJ() > d2.getJ() ) returnVal = 1;
        }
        return returnVal;
    }
}
```

Result, AJ1910.java

```
Data10:6,-3  Data10:6,55  Data10:20,82  Data10:20,94
```

1. The Comparator interface enables you to provide the methods compare and equals for comparison of objects whether or not their class implements Comparable. A Comparator reference can be passed to the Collections.sort method.

2. The Comparator interface specifies many methods, including:

    a. public int compare(Object o1, Object o2); Compare o1 and o2. Throw an Exception if o1's and o2's class types prevent them from being compared, or return an int:

        1) negative integer if o1 is less than o2
        2) zero if o1 is equal to o2
        3) positive integer if o1 is greater than o2

    b. public boolean equals(Object o); Return true if this object is equal to o.

        1) This method's return value should be consistent with that of the compare method. The equals method of Object is inherited by every class and can supply this method, but it returns true only if this object and o are the same object.
        2) A Comparator's compare and equals methods should find that the same objects are equal.

SORT LINKEDLIST VIA Comparable, EXAMPLE

AJ1912.java

```
import java.util.LinkedList;
import java.util.Collections;
public class AJ1912 {
    public static void main (String[] args) {

        LinkedList<Data12> list = new LinkedList<Data12>();
        list.add (new Data12 (20, 94));
        list.add (new Data12 ( 6, 55));
        list.add (new Data12 (20, 82));
        list.add (new Data12 ( 6, -3));

        Collections.sort (list);

        for (Object o : list)
            System.out.print (o + "  ");
        System.out.println ();
    }
}
```

Data12.java

```
import java.lang.Comparable;
public class Data12 implements Comparable<Data12> {
    private int i;
    private int j;

    public Data12 () {
    }
    public Data12 (int i, int j) {
       this.i = i;
       this.j = j;
    }
    public int getI () {
        return i;
    }
    public int getJ () {
        return j;
    }
    public String toString () {
        return "Data12:" + i + "," + j;
    }
    public boolean equals (Object o) {
        if (o instanceof Data12
        && this.i == ((Data12)o).getI()
        && this.j == ((Data12)o).getJ())
            return true;
        else
            return false;
    }
```

```

    /**sort ascending i as field 1, ascending j as field 2
    */
    public int compareTo (Data12 d)
    throws ClassCastException {

        int returnVal = 0;

        if (getI() < d.getI() ) returnVal = -1;
        if (getI() > d.getI() ) returnVal = 1;

        if (getI() == d.getI() ) {
            if (getJ() < d.getJ() ) returnVal = -1;
            if (getJ() == d.getJ() ) returnVal = 0;
            if (getJ() > d.getJ() ) returnVal = 1;
        }
        return returnVal;
    }
}
```

Result, AJ1912.java

```
Data12:6,-3  Data12:6,55  Data12:20,82  Data12:20,94
```

---

1. The Comparable interface in java.lang specifies one method:

    a. public int compareTo(Object o); Compare this object to o. Throw an Exception if this object's and o's class types prevent them from being compared, or return an int:

        1) negative integer if this object is less than o
        2) zero if this object is equal to o
        3) positive integer if this object is greater than o

2. To compare floats or doubles, you can use:

    a. Float.compare(float, float)
    b. Double.compare(double, double)

3. To compare Strings, such as book titles, and put nulls at the end of the sorted sequence:

```
if (otherRef == null) {return -1;}
if (otherRef.getTitle() == null) {return -1;}
if (title == null) {return 1;}
return title.compareTo(otherRef.getTitle() );
```

COMPARABLE AND COMPARATOR WITH STRINGS, EXAMPLE

AJ1914.java

```java
import java.util.*;
public class AJ1914 {
    public static void main (String[] args) {

        List<ClientName> list = new ArrayList<ClientName>();
        list.add (new ClientName("Arthur", "Zatch"));
        list.add (new ClientName("Marlene", "Sislert"));

        Collections.sort (list);
        for (ClientName cn : list) {
            System.out.println ("1. " + cn);
        }

        ComparatorLastNameAscend clna =
            new ComparatorLastNameAscend ();
        Collections.sort (list, clna);
        for (ClientName cn : list) {
            System.out.println ("2. " + cn);
        }
    }
}
```

ComparatorLastNameAscend.java

```java
import java.util.Comparator;
public class ComparatorLastNameAscend implements Comparator {

    public int compare (Object o1, Object o2)
    throws ClassCastException {
        if ( !(o1 instanceof ClientName)
          || !(o2 instanceof ClientName)) {
                throw new ClassCastException ();
        }

        String fn1 = ((ClientName)o1).getFirstName();
        String ln1 = ((ClientName)o1).getLastName();
        String fn2 = ((ClientName)o2).getFirstName();
        String ln2 = ((ClientName)o2).getLastName();

        if (ln1.compareTo(ln2) == 0) {  //ascending by
            return fn1.compareTo(fn2);  //lastname, firstname
        } else {
            return ln1.compareTo(ln2);
        }
    }
}
```

ClientName.java

```
public class ClientName implements Comparable {
    private String firstName = null;
    private String lastName  = null;

    public ClientName (String fn, String ln) {
        firstName = fn;
        lastName  = ln;
    }
    public String getFirstName () {
        return firstName;
    }
    public String getLastName () {
        return lastName;
    }
    public String toString () {
        return "ClientName:" + firstName + "," + lastName;
    }

    public int compareTo (Object o)  //ascending by
    throws ClassCastException {      //firstname, lastname
        if (! (o instanceof ClientName)) {
           throw new ClassCastException();
        }
        String fn = ((ClientName)o).getFirstName();
        String ln = ((ClientName)o).getLastName();

        if (firstName.compareTo(fn) == 0) {
            return lastName.compareTo(ln);
        } else {
            return firstName.compareTo(fn);
        }
    }
}
```

Result, AJ1914.java

```
1. ClientName:Arthur,Zatch
1. ClientName:Marlene,Sislert
2. ClientName:Marlene,Sislert
2. ClientName:Arthur,Zatch
```


OPTIONAL: Arrays CLASS, Arrays.sort() AND Arrays.binarySearch

1. The Arrays class in java.util, introduced in the Collections Framework in Java 1.2, contains public static methods for sorting and searching arrays. The Arrays class has no public constructor; objects of Arrays are not created.

2. The Arrays.binarySearch method is overloaded for use with arrays of many data types, so you can perform searches for a specified value using a binary search algorithm.

3. The Arrays.sort method is overloaded for use with arrays of Objects as well as all basic data types except boolean.

   a. For arrays of basic data types:

      1) public static void sort(datatype[] a); Sort the array into ascending order.

      ```
      Arrays.sort (a1);
      ```

      2) public static void sort(datatype[] a, int fromIndex, int toIndex); Sort elements in the specified range (including fromIndex, not including toIndex) into ascending order.

      ```
      Arrays.sort (a2, 2, 4);   //sorts for(i=2; i<4; i++)
      ```

   b. For arrays of references to objects whose class implements the Comparable interface and have a compareTo method:

      1) static void sort(Object[] a); Sort the elements into ascending order.

      2) static void sort(Object[] a, int fromIndex, int toIndex); Sort the elements in the specified range (including fromIndex, not including toIndex) into ascending order.

   c. For arrays of references to objects whose class has an associated class that implements the Comparator interface and has a compare method.

      1) static void sort(Object[] a, Comparator c); Sort the elements into the order created by Comparator c.

      2) static void sort(Object[] a, int fromIndex, int toIndex, Comparator c); Sort the elements in the range into the order created by Comparator c.

EXERCISES

1. Copy CaseStudy6.java and FoodVendor6.java, and call the copies CaseStudy19.java and FoodVendor19.java.

   FoodVendor19.java in com.themisinc.u19

   a. Create the methods listed below. If you are using Eclipse, view the different versions that Eclipse can generate for these methods.

      1) compareTo to sort ascending by contact
      2) toString

   b. Create an implements clause for Comparable.

   VendorDescendComparator.java in com.themisinc.u19

   c. Create a class called VendorDescendComparator.java that implements Comparator via a compare method, and compares FoodVendor19 objects for sorting by

      1) Major key, descending, companyName
      2) Minor key, descending, contact

   CaseStudy19.java in com.themisinc.u19

   d. Replace the array with an ArrayList. Use generics to ensure that the ArrayList contains only FoodVendor19 type.

   e. Add four elements of FoodVendor19 type to the ArrayList. Give two elements the same company name with different contacts, such as "Eben Food Corp." with contacts Karl Lenn and Inez Jonnet.

   f. Sort the ArrayList via the compare method of the VendorDescendComparator. Then use a loop to call the toString method of each FoodVendor19 to print a list of vendors and contacts in descending sequence.

   g. Sort the ArrayList via the compareTo method in FoodVendor19 and then use a loop to call the getContact method of each object and print a list of contacts in ascending sequence.

SOLUTIONS

VendorDescendComparator.java in com.themisinc.u19

```
package com.themisinc.u19;
import java.util.Comparator;

/**
*  Sort FoodVendor19 objects.
*  Major key, descending, companyName
*  Minor key, descending, contact
*/

public class VendorDescendComparator
  implements Comparator<FoodVendor19> {

    public int compare (
      FoodVendor19 fv1,
      FoodVendor19 fv2 )
      throws ClassCastException {

        //get fv1's companyName String,
        //then call that String's compareTo method
        //and pass fv2's companyName String to be compared

        int ret = fv1.getCompanyName().compareTo(
            fv2.getCompanyName() );

        //if the Strings are not equal,
        //reverse the positive-negative sign
        //and return that number to get a descending sort

        if (ret != 0) {
            return ret * -1;  //multiply by -1 to reverse
        }                     //1 to -1, or -1 to 1

        //the company names are the same, so compare the
        //minor key, contact, using the String method
        //compareTo

        ret = fv1.getContact().compareTo( fv2.getContact() );

        return ret * -1;      //multiply by -1 to reverse
                              //1 to -1, or -1 to 1
    }
}
```


FoodVendor19.java in com.themisinc.u19

```java
package com.themisinc.u19;
import java.io.Serializable;

public class FoodVendor19
    implements Serializable, Comparable<FoodVendor19> {

    private String companyName;
    private String contact;

    public FoodVendor19 (String companyName, String contact){
        setCompanyName (companyName);
        setContact (contact);
    }

    public String getCompanyName () {
        return companyName;
    }
    public void setCompanyName (String companyName) {
        this.companyName = companyName;
    }

    public String getContact () {
        return contact;
    }
    public void setContact (String contact) {
        this.contact = contact;
    }

    @Override
    public String toString () {
        return "FoodVendor19:" + companyName + "," + contact;
    }

    @Override
    public int compareTo (FoodVendor19 fv)
    throws ClassCastException {

        //the instance variable contact is a String,
        //so we can use the String class's compareTo method

        return contact.compareTo ( fv.getContact() );
    }
}
```


CaseStudy19.java in com.themisinc.u19

```
package com.themisinc.u19;
import java.util.ArrayList;
import java.util.Collections;

public class CaseStudy19 {
    public static void main (String[] args)
    throws CloneNotSupportedException {

        ArrayList<FoodVendor19> a =
            new ArrayList<FoodVendor19> ();

        a.add (new FoodVendor19 (
               "AB Food Services", "Arlene Banner") );
        a.add (new FoodVendor19 (
               "CD Foods, Inc", "Charles Denrick") );
        a.add (new FoodVendor19 (
               "Eben Food Corp.", "Karl Lenn") );
        a.add (new FoodVendor19 (
               "Eben Food Corp.", "Inez Jonnet") );

        VendorDescendComparator c =
            new VendorDescendComparator();
        Collections.sort (a, c);

        for (FoodVendor19 elem : a) {
            System.out.println (elem);
        }
        System.out.println ();

        Collections.sort (a);
        for (FoodVendor19 elem : a) {
            System.out.println (elem);
        }
    }
}
```

Result, CaseStudy19.java in com.themisinc.u19

```
FoodVendor19:Eben Food Corp.,Karl Lenn
FoodVendor19:Eben Food Corp.,Inez Jonnet
FoodVendor19:CD Foods, Inc,Charles Denrick
FoodVendor19:AB Food Services,Arlene Banner

FoodVendor19:AB Food Services,Arlene Banner
FoodVendor19:CD Foods, Inc,Charles Denrick
FoodVendor19:Eben Food Corp.,Inez Jonnet
FoodVendor19:Eben Food Corp.,Karl Lenn
```

UNIT 20: JAVA 8 NEW FEATURES: Optional CLASS, FUNCTIONAL INTERFACES, LAMBDA EXPRESSIONS, STREAM API

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

1. Use the Optional class to avoid NullPointerException and specify default values.

2. Briefly explain what a following features are, and recognize use of them in Java 1.8 programs.

    Functional interfaces
    Lambda expressions
    Method references
    Streams and the Stream API

java.util.Optional<T>

1. Optional is a final class and a container. An Optional object may contain a non-null value or no value.

2. Before Java 1.8, many methods were coded to return a null reference to indicate that an object was not present to be pointed to. This created the risk of a NullPointerException.

    a. The Optional class provides alternate way to indicate the absence of an object, called the absence of a value.

    b. If a method is coded to return an Optional reference, you know that the method may or may not return a non-null value.

3. Instance methods include:

    a. isPresent() If a value is present, return true.
    b. orElse(T default) Return the value, or if none, default.
    c. get() Return the non-null value held by this object, or throw NoSuchElementException if no value is present.

4. Static methods include:

    a. empty() Returns an empty Optional with no value present.
    b. of(T val) Returns an Optional with the non-null val or throws NullPointerException if val is null.
    c. ofNullable(T val) Returns an Optional holding val, if non-null, or an empty Optional.

5. The Optional class is also used in combination with lambda expressions, method references, and the Java 8 Stream API.

Optional CLASS, EXAMPLE

<u>AJ2003.java</u>

```
import java.util.Optional;
public class AJ2003 {
    public static void main(String[] args) {

/*1*/   //Create empty Optional object
        Optional<String> noValue = Optional.empty();
        if (noValue.isPresent() ) {
        } else {
             prin("1. noValue has no value");
        }

/*2*/   Optional<String> paris = Optional.of("Paris");
        prin("2. " + paris.orElse("Moscow") +
            ", noValue again=" + noValue.orElse("Rome") );

/*3*/   Integer i1 = null;
        Integer i2 = new Integer (22);
        Optional<Integer> o1 = Optional.ofNullable(i1);
        Optional<Integer> o2 = Optional.of(i2);
        prin( "3. " + add(o1,o2) );
    }

    public static Integer add (
        Optional<Integer> one, Optional <Integer> two) {
        prin("A. one=" + one.isPresent() );
        prin("B. two=" + two.isPresent() );
        return one.orElse(new Integer(11)) + two.get();
    }

    public static void prin (Object o) {
        System.out.println(o);
    }
}
```

<u>Result, AJ2003.java</u>

```
1. noValue has no value
2. Paris, noValue again=Rome
A. one=false
B. two=true
3. 33
```


FUNCTIONAL INTERFACES

1. A functional interface is an interface that has exactly one abstract method. This method normally specifies the single intended purpose of the interface which is one action. Two examples are the Comparable and Comparator interfaces.

   a. Functional interfaces may be called SAM types ("Single Abstract Method") or SMI types ("Single Method Interface.")

   b. Functional interfaces may also specify any public method defined in the Object class such as equals(), because Object's methods are inherited by all classes, and thus would be present in any class that implements the functional interface.

   c. Functional interfaces may also contain default and static methods, which were both introduced in Java 1.8.

2. The annotation @FunctionalInterface may be specified immediately above the header of a functional interface so the compiler can assure that it has only one abstract method.

3. Functional interfaces are closely associated with lambda expressions.

REVIEW

4. Java 1.8 interfaces may contain:

   a. Default methods which consist of concrete methods that an implementing class can override or use "as is."

   b. Static methods, which are default methods but the keyword default is not coded, only the keyword static.

      1) An implementing class cannot override a static method.

      2) In the implementing class, the static method name must be qualified by the interface name.

FUNCTIONAL INTERFACE, EXAMPLE

Interface2005.java

```
import java.util.Date;
@FunctionalInterface
public interface Interface2005 {

    int getInt (int i1, int i2) ;     //one abstract method

    boolean equals (Object o) ;           //in Object class

    public default Date getToday () {      //default method
        return new Date();
    }

    public static Date getTomorrow() {      //static method
        Date today = new Date();
        long t = today.getTime() + (1000 * 24 * 60 * 60);
        return new Date(t);
    }
}
```

AJ2005.java

```
public class AJ2005 implements Interface2005 {
    public static void main (String[] args) {

        AJ2005 ref = new AJ2005 ();

        int result = ref.getInt(2, 5);

        System.out.println ("result=" + result +
            "\ntoday=" + ref.getToday() +
            "\ntomorrow=" + Interface2005.getTomorrow() );
    }
    public int getInt (int i1, int i2) {
        return i1 + i2;
    }
}
```

Result, AJ2005.java

```
result=7
today=Thu Aug 17 21:01:40 EDT 2017
tomorrow=Fri Aug 18 21:01:40 EDT 2017
```


LAMBDA EXPRESSIONS WITH EXPRESSION OR BLOCK BODIES

1. A lambda expression is an unnamed "anonymous" method used to implement a method defined in a functional interface. When the lambda is assigned to a reference of the interface type, a class instance is constructed in which the interface's method is implemented via code in the lambda, thus creating a kind of anonymous class. Lambdas are also called closures.

2. Lambdas use an operator -> that was added to Java in version 1.8, called the lambda operator or arrow operator. The -> can be read as "becomes" or "goes to."

   a. On the left you specify the lambda's parameters. The parentheses may be empty, may contain one parameter, or may contain multiple comma-separated parameters. Parentheses are required only for multiple parameters.

   b. On the right you specify the lambda's procedure. If it is one simple expression ending with ; semicolon, the lambda is called an expression lambda. If the procedure is a block in { } curly braces, the lambda is called a block lambda.

3. Block lambdas can receive parameters, throw Exceptions and must have a return statement if they return a value.

4. Lambdas are a concise way to implement a functional interface and eliminate the need to define a new class or create an anonymous inner class to implement a single method.

   a. A lambda is a piece of code that can be referenced and passed to another piece of code for execution.

   b. Lambdas and the Stream API, introduced later in this unit, enhance the capabilities of the Collections Framework.

5. Examples:

   a. `() -> 1.2;//No args, same as:  double meth(){return 1.2;}`

   b. `(i, j) -> i + j;`

   c. `(int i, int j) -> i + j;`

EXPRESSION LAMBDAS, BLOCK LAMBDA, EXAMPLE

<u>AJ2007.java</u>

```
import java.util.Date;
interface Inter7Long {
    long getLong () ;
}
interface Inter7Double {
    double getDouble (double d1, double d2) ;
}
public class AJ2007 {
    public static void main(String[] args) {

/*1*/   Inter7Long refL;
        refL = () -> 12L;
        System.out.println("1. " + refL.getLong() );

/*2*/   refL = () -> new Date().getTime();
        System.out.println("2. " + refL.getLong() );

/*3*/   //String is not compatible with long
        //refL = () -> new String ("123.45");

/*4*/   Inter7Double refD = (d1, d2) -> (d1 + d2);
        System.out.println("3. " + refD.getDouble(1.2, 3.4));

/*5*/   refD = (double one, double two) -> {
             double result = one * two;
             if ( result > 256.00 ) {
                 return 0.0;
             } else {
                 return result;
             }
        };
        System.out.println("4. " + refD.getDouble(1.2, 3.4));
    }
}
```

<u>Result, AJ2007.java</u>

```
1. 12
2. 1503182347465
3. 4.6
4. 4.08
```


LAMBDA EXPRESSION INSTEAD OF A Comparator CLASS

AJ2008.java

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
public class AJ2008 {
    public static void main (String[] args) {

/*1*/   ArrayList<Integer> a = new ArrayList<Integer>();
        a.add (new Integer(20) );
        a.add (new Integer(6)  );
        a.add (new Integer(82) );
        a.add (new Integer(-3) );

/*2*/   Collections.sort(a,(Integer i1,Integer i2) -> i1-i2);

        for (Object o : a)
            System.out.print (o + "  ");
        System.out.println ();

/*3*/   Comparator<Integer> comp =
            ( (Integer i1, Integer i2) -> i2 - i1 );

        Collections.sort (a, comp);       //see page aj19.10

        for (Object o : a)
            System.out.print (o + "  ");
        System.out.println ();
    }
}
```

Result, AJ2008

```
-3  6  20  82
82  20  6  -3
```

1. Starting in Java 1.8, methods can receive lambdas as parameters, if the method parameter is specified as the functional interface type.

2. The lambda is treated as a special implementation of a functional interface. Java creates an object as if you had used the new operator with the name of class's constructor.

METHOD REFERENCES

AJ2009.java

```
import java.util.Collections;
import java.util.List;
import java.util.ArrayList;
public class AJ2009 {

   public static void main(String[] args) {
      List<String> a = new ArrayList<String>();
      a.add("Amy");
      a.add("Walter");
      a.add("Marion");

/*1*/ Collections.sort(a,
        (String s1,String s2)->AJ2009.compareStrings(s1,s2));
      System.out.println("1. " + a);

/*2*/ Collections.sort (a, Collections.reverseOrder());
      System.out.println("2. " + a);

/*3*/ Collections.sort(a, AJ2009::compareStrings);
      System.out.println("3. " + a);
   }
   public static int compareStrings (String s1, String s2) {
      return s1.compareTo(s2);
   }
}
```

Result, AJ2009.java

```
1. [Amy, Marion, Walter]
2. [Walter, Marion, Amy]
3. [Amy, Marion, Walter]
```

===

1. Method references use :: double colon and can point to methods as follows:

	Kind of Method	Reference syntax
a.	static methods	ClassName::methodName
b.	instance method	ObjectReferenceName::methodName
c.	constructors	ClassName::new

INTRODUCTION TO Java 1.8 STREAMS

1. Java 1.8 Streams use generics, lambdas, and method references to let you specify operations on elements from collections, such as search, filter (select elements based on specified criteria), count, sort, map elements to different class types, etc.

    a. Stream actions conceptually resemble pipelines or database queries.

    b. A stream is a series of elements that are passed through a series of operations.

2. The package java.util.stream contains the interfaces that support streams. The top stream interface is BaseStream. The most commonly used methods are defined in the interface Stream.

3. Methods may be intermediate or terminal.

    a. An intermediate operation produces another stream, and can be chained to another method to create a pipeline. Intermediate operations are called "lazy" because their action is performed when the terminal action is done, thus they are efficient. Examples: filter, map, mapToInt.

    b. A terminal operation consumes the stream as it produces a result, so it cannot be chained to another method. Examples: count, forEach, max, min.

Result, StreamDemo.java

```
1. mylist=[44, 12, -9, -76, 35]
2. mystream=java.util.stream.ReferencePipeline$Head@106d69c
3. min=Optional[-76]
4. max=44
5. mystream after sorted()=java.util.stream.SortedOps$OfRef@9e54c2
6. sorted stream forEach=-76, -9, 12, 35, 44,
7. one filter followed by forEach=12, 35, 44,
8. two filters followed by forEach=12, 35,
9. total of positive numbers=91
```


StreamDemo.java

```
import java.util.ArrayList;
import java.util.List;
import java.util.Optional;
import java.util.stream.Stream;
public class StreamDemo {
    public static void main(String[] args) {
/*1*/   List<Integer> mylist = new ArrayList<> ();
        mylist.add (new Integer (44));
        mylist.add (new Integer (12));
        mylist.add (new Integer (-9));
        mylist.add (new Integer (-76));
        mylist.add (new Integer (35));
        p ("1. mylist=" + mylist + "\n");

/*2*/   Stream<Integer> mystream = mylist.stream();
        p ("2. mystream=" + mystream + "\n");

/*3*/   Optional<Integer> minI = mystream.min(Integer::compare);
        p ("3. min=" + (minI.isPresent() ? minI : "none") + "\n");

/*4*/   mystream = mylist.stream();
        Optional<Integer> maxI = mystream.max(Integer::compare);
        if (maxI.isPresent()) p ("4. max=" + maxI.get() + "\n");

/*5*/   mystream = mylist.stream().sorted();
        p ("5. mystream after sorted()=" + mystream + "\n");

/*6*/   p ("6. sorted stream forEach=");
        mystream.forEach((n) -> p (n + ", "));
        p ("\n");

/*7*/   mystream = mylist.stream().sorted().filter((n) -> n>0);
        p ("7. one filter followed by forEach=");
        mystream.forEach((n) -> p (n + ", "));
        p ("\n");

/*8*/   mystream = mylist.stream().sorted()
            .filter( (n) -> n>0 )
            .filter( (n) -> n<40 );
        p ("8. two filters followed by forEach=");
        mystream.forEach((n) -> p (n + ", "));
        p ("\n");

/*9*/   int total = mylist.stream()
            .filter( n -> n>0 ).mapToInt( n -> n ).sum();
        p ("9. total of positive numbers=" + total);
    }
    public static void p (String s) {System.out.print (s);}
}
```


(blank)

APPENDIX 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  YourClassname.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:\jdk1.8\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:\jdk1.8\bin`

   b. UNIX sh or ksh `PATH=$PATH:/jdk1.8/bin  ;  export PATH`

   c. UNIX csh `setenv  PATH  ${PATH}:/jdk1.8/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

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 bin,
            com                       src, 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. src folder: your source files will go under src, usually in a subfolder specified in the package statement.

   f. test folder: if you use test drivers such as JUnit files, you would typically put them in a folder called test.

   g. .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. <u>Highlight all occurrences of an identifier</u>

    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. <u>Highlight from open to close { } or [ ] or ( )</u>

    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. <u>Highlight and copy a line in the editor</u>

    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. <u>Overriding methods indicated by Annotation</u>

    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.

NEW CLASS IN NEW PACKAGE, RENAME, CONSOLE VIEW, PROBLEMS VIEW

1. Create a new 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. More on packages: see page E.25.

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 with 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. <u>Pass commandline arguments to main</u>

    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. <u>Display 2 files in side-by-side Editors</u>

    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. <u>Navigator view, Comparison to Package Explorer</u>

    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. <u>Display file properties</u>

    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. <u>Overview</u>

   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. <u>Export files and/or a folder tree into a jar file</u>

   a. Click File, Export. In the "Export" popup expand Java. Highlight "JAR file". Click Next>.

   b. In the "Jar Export" popup, under "Select the resources to export:" in the large box on the LEFT select the projects and/or directories and/or files to be exported.

   c. Select the content you want to export in a jar file. Your choices are:

      1) "Export generated class files and resources"
      2) "Export all output folders for checked projects"
      3) "Export Java source files and resources"

      NOTE: Eclipse rebuilds class files when it imports. If jar files will be used only to import source files into Eclipse, do not put .class files in the jar.

   d. Under "Select the export destination:" fill in the full pathname for your jar file with the .jar extension. For example: C:\myjava\myJar.jar

   e. Under "Options:" select the checkboxes:

      1) "Compress the contents of the JAR file" if you want compression.
      2) Select the checkbox for "Add directory entries".

   f. Decide whether you want to click the checkbox for "Overwrite existing files without warning".

   g. Click Next>, Finish.

JAR FILES: VIEW CONTENTS, IMPORT, JRE System Library

3. View contents of a jar file

    a. Via a commandline: see Unit 18.
    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, highlight the src folder. 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". Click Finish.
    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. Import a new project from a jar into Eclipse

    a. Click File, Import. In the "Import Select" popup, expand the "General" section. Highlight "Existing Projects Into Workspace". Click Next>.
    b. In the "Import Projects" popup, click "Select Archive File"
    c. Browse to the directory with your jar. Highlight the jar. Click Open. The name of the project in the jar will be displayed and selected in the big "Projects:" box. Click Finish.

6. 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. Alignment of Curly Braces

   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. Editor Font and Point Size

   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 <u>index.html</u>. Your javadocs can be placed in a central location on a server so they are available to the development team.

   g. <u>Link your project javadocs to the JDK javadocs</u> 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.
    a. In the popup click New, Source Folder.
    b. 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. 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.

MOVE PACKAGE
LET Project1 IMPORT CLASSES FROM Project2
SPECIFY A PACKAGE AFTER CREATING A SOURCE FILE

MOVE A PACKAGE

1. To 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.

LET Project1 IMPORT classes FROM Project2

1. In Package Explorer highlight Project1, then right click. Click Properties. In the list on the left click "Java Build Path". In the large box on the right click the Projects tab. Click Add. In the popup with the title "Required Project Selection" click checkbox for Project2. Click OK, Apply, OK.

    a. This makes Project2 classes available to Project1. If one or more classes in Project2 have the same classname and package as classes in Project1, you will be notified via a popup. If you click "Continue" then in the build path Project2 will be added AFTER Project1, and when you use a class that has the same classname and package in both projects, you will always get the class in Project1.

    c. Using the default package in Eclipse is not recommended. Classes in the default package cannot be imported, whether in the same or a different project.

SPECIFY A PACKAGE AFTER CREATING A SOURCE FILE

1. Source files that were created without a package will be in the project's src folder. Before you can specify a package for these files, that package must exist.

    a. <u>Make a new package</u>. If the package does not exist yet: highlight the project in which you want the new package. Right click, click New, Package. In the "New Java Package" popup, fill in the name for the new package. Click Finish.

2. Three ways to move a class to an existing package.

    a. Expand the folder trees in Package Explorer so that the display shows both the folder where the classes are and the folder where you want to move them. Drag the source file to the desired package name and drop it. A "Move" popup will appear with a check in the option "Update references to 'YourClassName.java'. Click OK.

b. Highlight the source file to be moved in Package Explorer. Right click on it, click Refactor, Move. (Alternatively, after highlighting the source file in Package Explorer, click Refactor in the Eclipse menu bar, and then click Move.) In the "Move" popup highlight the desired destination folder. There will be a check in the option "Update references to 'YourClassName.java'. Click OK. The package statement in the source file will be updated automatically.

c. If the source file already has a package statement, bring the source file to be moved into the Editor. Change the name of the package. This causes an error. Hover your mouse over the red error indicator and right click. In the popup click Quick Fix. Double click on Move 'YourClassName.java' to 'destination'.

ECLIPSE EXERCISE AFTER UNIT 4 (based on Kepler 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. Extract All from the zip download and accept the default directory name eclipse-jee-kepler-R-win32 and MAKE A NOTE OF WHERE THE DIRECTORY IS.
   b. Use Windows Explorer to go to eclipse-jee-kepler-R-win32. Go to the subdirectory eclipse. The icon for eclipse.exe is a blue sphere with four 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" in the top left.
   b. If the title bar says "Java EE", click Window, Open Perspective, Java.

7. Create your project J2unit04
   a. Click File, New, Java Project.
   b. In the "New Java Project" popup, enter J2unit04 for your project name, and Eclipse will make the folder for it.
   c. Click the square checkbox for "Use default location"
   d. Do not change the JRE. The top round radio button should be selected, which should be labeled "Use an execution environment JRE:" with selection "JavaSE-1.7".
   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. Click File, New, Class.
   b. For "Source folder" enter J2unit04/src
   c. For "Package:" enter com.themisinc.u04 and Eclipse will make the folders if they don't exist.
   d. For "Name:" enter the new class name RoomReservation4
   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. Type in three symbolic constants:

```
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;
```

10. Type in four declarations for input variables:

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

11. Type in one declaration for a calculated amount:

```
private double roomAmount;
```

12. Make Getters and setters.

    a. Place your cursor on the line before where you want the getters amd setters. Click Source, Generate Getters and Setters...

        1) Click the checkboxes for reservationNumber, seats, numberOfDays, and dayRatePerSeat;

        2) Click OK.

13. Modify the setSeats method to validate the seats variable via a switch to ensure that the value is 10 or 12 or 14.

    a. Place your cursor inside the setSeats method where you want your switch, BEFORE the line this.seats = seats;

    b. Type the characters sw and then press Control-Space. A popup will show you a choice of the switch structure. Double-click to select the switch.

    c. The search expression in () parentheses should be seats so that the line says switch (seats) {

    d. Instead of typing System.err.println, type syserr and press Control-Space.

    e. To complete the method, use the solution in Unit 4 as a guide.

14. Modify the setNumberOfDays method to validate the numberOfDays variable via an if structure to ensure that the value is in the range 1 through 5.

    a. Place your cursor inside the setNumberOfDays method where you want your if BEFORE the line this.name = name;

    b. Type the two characters if and then press Control-Space. A popup will show you a choice of if structures. Double-click to select the kind of if you want.

    c. Fill in your boolean expression:

    ```
    if (numberOfDays < 1 || numberOfDays > 5)
    ```

    d. To complete the method, use the solution in Unit 4 as a guide.

15. Modify the setDayRatePerSeat method to validate the dayRatePerSeat variable via an if structure to ensure that the value is in the range 25.00 through 65.00. Use the procedure in step 14 above as a guide.

16. Make your constructors.

    a. Null constructor: place your cursor on the line before where you want the null contructor. Click Source, Generate Constructor using Fields..., Deselect All, OK.

    b. Constructor with four parameters: place your cursor on the line before where you want the contructor. Click Source, Generate Constructor using Fields.... Click the checkboxes for reservationNumber, seats, numberOfDays, and dayRatePerSeat. Click OK.

        1) Modify the assignments in the constructor to call your set methods. For example, Eclipse gives you this.seats = seats; Change it to setSeats (seats);

    c. Constructor with three parameters: place your cursor on the line before where you want the contructor. Click Source, Generate Constructor using Fields.... Click the checkboxes for reservationNumber, seats, and numberOfDays. Click OK.

        1) Modify the constructor to call this and pass the parameters reservationNumber, seats, numberOfDays, and DEFAULT_DAY_RATE_PER_SEAT.

17. Make your calculateAmount method by typing it in. Use the solution in Unit 4 as a guide.

18. Make your printOneReservation method by typing it in. Use the solution in Unit 4 as a guide.

    a. Instead of typing System.out.println, type sysout and press Control-Space.

19. 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.

20. Create a main class. Use the procedure in paragraph 8, but use class name CaseStudy4.java and BEFORE YOU CLICK "Finish" CLICK THE CHECKBOX FOR A main METHOD. You can delete the comment template in main.

21. In the main method, create two RoomReservation4 objects.

    a. To avoid typing the whole name RoomReservation4, type Ro and then press Control-Space. Double-click to choose RoomReservation4 from the popup box.

    b. Use the solution in Unit 4 as a guide to the parameters to pass to the RoomReservation4 constructors.

22. Call the printOneReservation method of each object to print the data in the object.

    a. In the parentheses of System.out.println, type the name of a RoomReservation4 reference and a dot character, and then pause after the dot until the template box pops up with choices of methods to be called. You may have to press Control-Space two or three times to see all methods to choose from. Select the printOneReservation method of RoomReservation4.

    b. An alternative way to call the method is to type the name of the reference, the dot, and a few characters of the method name, such as rr1.pr and then press Control-Space. Eclipse will fill in the method name.

23. While the main class is displayed in the Editor, execute your program. The run icon is a green circle with white triangle.

    a. If you have not saved, the first time you click the run icon, a "Save and Launch" popup asks which resources to save. Select your files.

        1) Click the checkbox for "Always save resources before launching" to make Eclipse save automatically before compiling and executing.
        2) The Editor allows you to undo via CTRL-Z after you save, compile, and find errors.

    b. Click OK.

    c. The Console view below the Editor will display output from System.out.println in black, and output from System.err.println in red.

YOU DID IT! Now, you will want to know more about Eclipse!

24. The Outline view on the right side of the Eclipse screen shows you the members of the class displayed in the Editor. 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 folders under com.themisinc.u04.

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.

28. To change the font in the Editor, click Window, Preferences. In the Preferences popup, in the panel on the left, expand the subtopics under General. Then expand the subtopics under Appearance. 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. Courier New is available in many sizes. Select your choice. Click OK, OK.