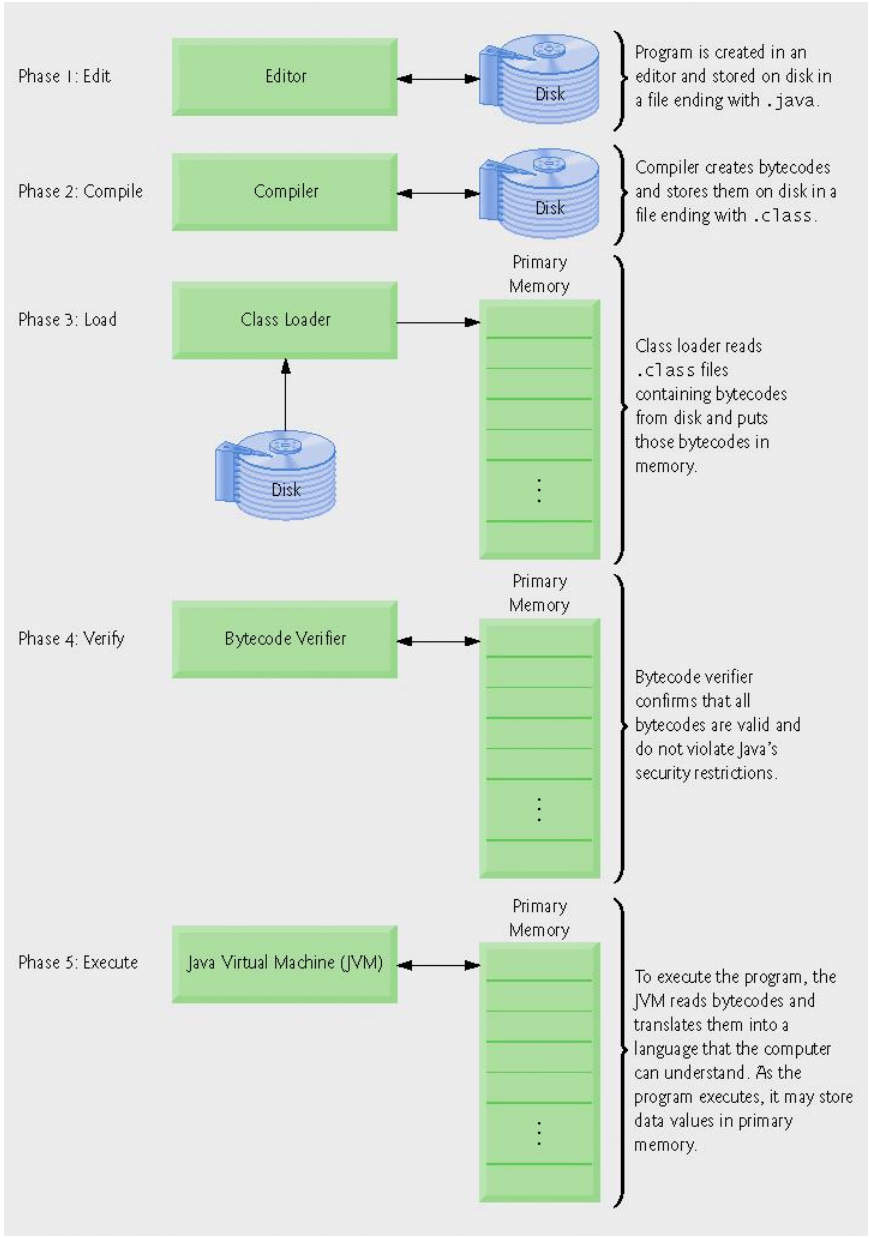
What is Java?

- Java
  - Originally for intelligent consumer-electronic devices
  - Then used for creating Web pages with dynamic content
  - Now also used to:
    - Develop large-scale enterprise applications
    - Enhance WWW server functionality
    - Provide applications for consumer devices (cell phones, etc.)

1.13 Typical Java Development Environment

- Java programs normally undergo five phases
  - Edit
    - Programmer writes program (and stores program on disk)
  - Compile
    - Compiler creates bytecodes from program
  - Load
    - Class loader stores bytecodes in memory
  - Verify
    - Bytecode Verifier confirms bytecodes do not violate security restrictions
  - Execute
    - JVM translates bytecodes into machine language


Phase 1: Edit
Editor
Disk
Program is created in an editor and stored on disk in a file ending with .java.
Phase 2: Compile
Compiler
Disk
Compiler creates bytecodes and stores them on disk in a file ending with .class.
Phase 3: Load
Class Loader
Primary Memory
Disk
Class loader reads .class files containing bytecodes from disk and puts those bytecodes in memory.
Phase 4: Verify
Bytecode Verifier
Primary Memory
Bytecode verifier confirms that all bytecodes are valid and do not violate Java's security restrictions.
Phase 5: Execute
Java Virtual Machine (JVM)
Primary Memory
To execute the program, the JVM reads bytecodes and translates them into a language that the computer can understand. As the program executes, it may store data values in primary memory.

```java
// Fig. 2.1: Welcome1.java
// Text-printing program.

public class Welcome1
{
   // main method begins execution of Java application
   public static void main( String args[] )
   {
      System.out.println( "Welcome to Java Programming!" );

   } // end method main

} // end clazss Welcome1
```

```
Welcome to Java Programming!
```

Another program

```
1  // Fig. 2.7: Addition.java
2  // Addition program that displays the sum of two numbers.
3  import java.util.Scanner; // program uses class Scanner
4
5  public class Addition
6  {
7     // main method begins execution of Java application
8     public static void main( String args[] )
9     {
10       // create Scanner to obtain input from command window
11       Scanner input = new Scanner( System.in );
12
13       int number1; // first number to add
14       int number2; // second number to add
15       int sum; // sum of number1 and number2
16
17       System.out.print( "Enter first integer: " ); // prompt
18       number1 = input.nextInt(); // read first number from user
19
```

```
20        System.out.print( "Enter second integer: " ); // prompt
21        number2 = input.nextInt(); // read second number from user
22
23        sum = number1 + number2; // add numbers
24
25        System.out.printf( "Sum is %d\n", sum ); // display sum
26
27     } // end method main
28
29 } // end class Addition
```

```
Enter first integer: 45
Enter second integer: 72
Sum is 117
```

Arithmetic Operations

- Operators precedence and arithmetic operations are almost the same as C/C++
- If statements and conditions are exactly the same.

Classes

- Class definitions are very similar to C++
- Access modifiers (public, private, protected) are the same.
- Java class functions are called **methods**
- Java class data members are called **fields** or **instance variables**.
- No global functions or variables in Java. Everything is objects.

Classes (Cont.)

- Static methods are the same as static member functions.
- Static fields are the same as static data members.
- Static members are declared like in C++.
- Java uses static methods of Math class like global functions in C++.
- For constants, Java uses keyword **`final`**.
- Method main in an application class is **static**.

Primitive Types vs. Reference Types

- Types in Java
  - Primitive (their sizes are not machine dependent)
    - **`boolean`**, **`byte`**, **`char`**, **`short`**, **`int`**, **`long`**, **`float`**, **`double`**
  - Reference (sometimes called nonprimitive types)
    - Objects
    - Default value of **`null`**
    - Used to invoke an object's methods

```java
// Fig. 3.10: GradeBook.java
// GradeBook class with a constructor to initialize the course name.

public class GradeBook
{
   private String courseName; // course name for this GradeBook

   // constructor initializes courseName with String supplied as argument
   public GradeBook( String name )
   {
      courseName = name; // initializes courseName
   } // end constructor

   // method to set the course name
   public void setCourseName( String name )
   {
      courseName = name; // store the course name
   } // end method setCourseName

   // method to retrieve the course name
   public String getCourseName()
   {
      return courseName;
   } // end method getCourseName
```

```
25
26    // display a welcome message to the GradeBook user
27    public void displayMessage()
28    {
29       // this statement calls getCourseName to get the
30       // name of the course this GradeBook represents
31       System.out.printf( "Welcome to the grade book for\n%s!\n",
32          getCourseName() );
33    } // end method displayMessage
34
35 } // end class GradeBook
```

```java
// Fig. 3.11: GradeBookTest.java
// GradeBook constructor used to specify the course name at the
// time each GradeBook object is created.

public class GradeBookTest
{
   // main method begins program execution
   public static void main( String args[] )
   {
      // create GradeBook object
      GradeBook gradeBook1 = new GradeBook(
         "CS101 Introduction to Java Programming" );
      GradeBook gradeBook2 = new GradeBook(
         "CS102 Data Structures in Java" );

      // display initial value of courseName for each GradeBook
      System.out.printf( "gradeBook1 course name is: %s\n",
         gradeBook1.getCourseName() );
      System.out.printf( "gradeBook2 course name is: %s\n",
         gradeBook2.getCourseName() );
   } // end main

} // end class GradeBookTest
```

```
gradeBook1 course name is: CS101 Introduction to Java Programming
gradeBook2 course name is: CS102 Data Structures in Java
```

Control Statements

- All control statements (if, if-else, ?:, switch) are all the same.
- The repetition statements (while, for, do-while) are all the same.
- Compound assignments (+=, *=, etc.) and post-pre increment decrement (--, ++) exist in Java
- break and continue have the same effect.

Logical Operators

- Logical && and || have the same effect.
- Boolean Logical AND (&) Operator
  - Works identically to &&
  - Except & always evaluate both operands
- Boolean Logical OR (|) Operator
  - Works identically to ||
  - Except | always evaluate both operands

Promotions allowed for primitive types.

sizes are not machine dependent!!

Type	Valid promotions
double	None
float	double
long	float or double
int	long, float or double
char	int, long, float or double
short	int, long, float or double (but not char)
byte	short, int, long, float or double (but not char)
boolean	None (boolean values are not considered to be numbers in Java)

Java API Packages

- **No include statements in Java.**
- Including the declaration
  `import java.util.Scanner;`
  allows the programmer to use `Scanner` instead of `java.util.Scanner`
- Java API documentation JDK 7
  - https://docs.oracle.com/javase/7/docs/api/

Java API packages (a subset). (Part 1 of 2)

Package	Description
`java.applet`	The Java Applet Package contains a class and several interfaces required to create Java applets—programs that execute in Web browsers. (Applets are discussed in Chapter 20, Introduction to Java Applets; interfaces are discussed in Chapter 10, Object_-Oriented Programming: Polymorphism.)
`java.awt`	The Java Abstract Window Toolkit Package contains the classes and interfaces required to create and manipulate GUIs in Java 1.0 and 1.1. In current versions of Java, the Swing GUI components of the `javax.swing` packages are often used instead. (Some elements of the `java.awt` package are discussed in Chapter 11, GUI Components: Part 1, Chapter 12, Graphics and Java2D, and Chapter 22, GUI Components: Part 2.)
`java.awt.event`	The Java Abstract Window Toolkit Event Package contains classes and interfaces that enable event handling for GUI components in both the `java.awt` and `javax.swing` packages. (You will learn more about this package in Chapter 11, GUI Components: Part 1 and Chapter 22, GUI Components: Part 2.)
`java.io`	The Java Input/Output Package contains classes and interfaces that enable programs to input and output data. (You will learn more about this package in Chapter 14, Files and Streams.)
`java.lang`	The Java Language Package contains classes and interfaces (discussed throughout this text) that are required by many Java programs. This package is imported by the compiler into all programs, so the programmer does not need to do so.

Java API packages (a subset). (Part 2 of 2)

Package	Description
`java.net`	The Java Networking Package contains classes and interfaces that enable programs to communicate via computer networks like the Internet. (You will learn more about this in Chapter 24, Networking.)
`java.text`	The Java Text Package contains classes and interfaces that enable programs to manipulate numbers, dates, characters and strings. The package provides internationalization capabilities that enable a program to be customized to a specific locale (e.g., a program may display strings in different languages, based on the user's country).
`java.util`	The Java Utilities Package contains utility classes and interfaces that enable such actions as date and time manipulations, random-number processing (class `Random`), the storing and processing of large amounts of data and the breaking of strings into smaller pieces called tokens (class `StringTokenizer`). (You will learn more about the features of this package in Chapter 19, Collections.)
`javax.swing`	The Java Swing GUI Components Package contains classes and interfaces for Java's Swing GUI components that provide support for portable GUIs. (You will learn more about this package in Chapter 11, GUI Components: Part 1 and Chapter 22, GUI Components: Part 2.)
`javax.swing.event`	The Java Swing Event Package contains classes and interfaces that enable event handling (e.g., responding to button clicks) for GUI components in package `javax.swing`. (You will learn more about this package in Chapter 11, GUI Components: Part 1 and Chapter 22, GUI Components: Part 2.)

Scoping, shadowing, overloading

- They work the same way as in C++
- Methods can be overloaded.

```java
// Fig. 6.11: Scope.java
// Scope class demonstrates field and local variable scopes.

public class Scope
{
   // field that is accessible to all methods of this class
   private int x = 1;

   // method begin creates and initializes local variable x
   // and calls methods useLocalVariable and useField
   public void begin()
   {
      int x = 5; // method's local variable x shadows field x

      System.out.printf( "local x in method begin is %d\n", x );

      useLocalVariable(); // useLocalVariable has local x
      useField(); // useField uses class Scope's field x
      useLocalVariable(); // useLocalVariable reinitializes local x
      useField(); // class Scope's field x retains its value

```

```
22       System.out.printf( "\nlocal x in method begin is %d\n", x );
23    } // end method begin
24
25    // create and initialize local variable x during each call
26    public void useLocalVariable()
27    {
28       int x = 25; // initialized each time useLocalVariable is called
29
30       System.out.printf(
31          "\nlocal x on entering method useLocalVariable is %d\n", x );
32       ++x; // modifies this method's local variable x
33       System.out.printf(
34          "local x before exiting method useLocalVariable is %d\n", x );
35    } // end method useLocalVariable
36
37    // modify class Scope's field x during each call
38    public void useField()
39    {
40       System.out.printf(
41          "\nfield x on entering method useField is %d\n", x );
42       x *= 10; // modifies class Scope's field x
43       System.out.printf(
44          "field x before exiting method useField is %d\n", x );
45    } // end method useField
46 } // end class Scope
```

```java
// Fig. 6.12: ScopeTest.java
// Application to test class Scope.

public class ScopeTest
{
   // application starting point
   public static void main( String args[] )
   {
      Scope testScope = new Scope();
      testScope.begin();
   } // end main
} // end class ScopeTest
```

```
local x in method begin is 5

local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 1
field x before exiting method useField is 10

local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 10
field x before exiting method useField is 100

local x in method begin is 5
```

Arrays

- Arrays have similarities and differences from C++.
- Created dynamically with keyword new

```
int[] c = new int[ 12 ];
```

  - Equivalent to

```
int[] c;  // declare array variable
c = new int[ 12 ]; // create array
```

- They are not pointers. **They are reference types.**
- Array initializers can be used.
- Multiple dimensional arrays are possible.

```java
// Fig. 7.2: InitArray.java
// Creating an array.

public class InitArray
{
   public static void main( String args[] )
   {
      int array[]; // declare array named array

      array = new int[ 10 ]; // create the space for array

      System.out.printf( "%s%8s\n", "Index", "Value" ); // column headings

      // output each array element's value
      for ( int counter = 0; counter < array.length; counter++ )
         System.out.printf( "%5d%8d\n", counter, array[ counter ] );
   } // end main
} // end class InitArray
```

```
Index   Value
    0       0
    1       0
    2       0
    3       0
    4       0
    5       0
    6       0
    7       0
    8       0
    9       0
```

7.6 Enhanced `for` Statement

- Enhanced `for` statement Java SE 5
  - Allows iterates through elements of an array or a collection without using a counter
  - Syntax

```
for ( parameter : arrayName )
   statement
```

```java
// Fig. 7.12: EnhancedForTest.java
// Using enhanced for statement to total integers in an array.

public class EnhancedForTest
{
   public static void main( String args[] )
   {
      int array[] = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
      int total = 0;

      // add each element's value to total
      for ( int number : array )
         total += number;

      System.out.printf( "Total of array elements: %d\n", total );
   } // end main
} // end class EnhancedForTest
```

```
Total of array elements: 849
```

Local variable declarations with initializers:

Local variable declarations with initializers: **Java SE 10**

```
var list = new ArrayList<String>();    //infers ArrayList<String>
var stream = list.stream();            // infers Stream<String>
var path = Paths.get(fileName);        // infers Path
var bytes = Files.readAllBytes(path);  // infers bytes[]
```

Passing Data to Methods

- Notes on passing arguments to methods
  - Two ways to pass arguments to methods
    - Pass-by-value
      - Copy of argument's value is passed to called method
      - In Java, every primitive is pass-by-value
    - Pass-by-reference
      - Caller gives called method direct access to caller's data
      - Called method can manipulate this data
      - Improved performance over pass-by-value
      - In Java, every object is pass-by-reference
        - » In Java, arrays are objects
        - » Therefore, arrays are passed to methods by reference

Variable-Length Argument Lists

- Variable-length argument lists
  - New feature in J2SE 5.0
  - Unspecified number of arguments
  - Use ellipsis (…) in method's parameter list
    - Can occur only once in parameter list
    - Must be placed at the end of parameter list
  - Array whose elements are all of the same type

```java
// Fig. 7.20: VarargsTest.java
// Using variable-length argument lists.

public class VarargsTest
{
   // calculate average
   public static double average( double... numbers )
   {
      double total = 0.0; // initialize total

      // calculate total using the enhanced for statement
      for ( double d : numbers )
         total += d;

      return total / numbers.length;
   } // end method average

   public static void main( String args[] )
   {
      double d1 = 10.0;
      double d2 = 20.0;
      double d3 = 30.0;
      double d4 = 40.0;

```

```
25       System.out.printf( "d1 = %.1f\nd2 = %.1f\nd3 = %.1f\nd4 = %.1f\n\n",
26          d1, d2, d3, d4 );
27
28       System.out.printf( "Average of d1 and d2 is %.1f\n",
29          average( d1, d2 ) );
30       System.out.printf( "Average of d1, d2 and d3 is %.1f\n",
31          average( d1, d2, d3 ) );
32       System.out.printf( "Average of d1, d2, d3 and d4 is %.1f\n",
33          average( d1, d2, d3, d4 ) );
34    } // end main
35 } // end class VarargsTest
```

```
d1 = 10.0
d2 = 20.0
d3 = 30.0
d4 = 40.0

Average of d1 and d2 is 15.0
Average of d1, d2 and d3 is 20.0
Average of d1, d2, d3 and d4 is 25.0
```

Using Command-Line Arguments

- Command-line arguments
  - Pass arguments from the command line
    - `String args[]`
  - Appear after the class name in the `java` command
    - `java MyClass a b`
  - Number of arguments passed in from command line
    - `args.length`
  - First command-line argument
    - `args[ 0 ]`

```java
// Fig. 7.21: InitArray.java
// Using command-line arguments to initialize an array.

public class InitArray
{
   public static void main( String args[] )
   {
      // check number of command-line arguments
      if ( args.length != 3 )
         System.out.println(
            "Error: Please re-enter the entire command, including\n" +
            "an array size, initial value and increment." );
      else
      {
         // get array size from first command-line argument
         int arrayLength = Integer.parseInt( args[ 0 ] );
         int array[] = new int[ arrayLength ]; // create array

         // get initial value and increment from command-line argument
         int initialValue = Integer.parseInt( args[ 1 ] );
         int increment = Integer.parseInt( args[ 2 ] );

         // calculate value for each array element
         for ( int counter = 0; counter < array.length; counter++ )
            array[ counter ] = initialValue + increment * counter;

         System.out.printf( "%s%8s\n", "Index", "Value" );

```

```
29          // display array index and value
30          for ( int counter = 0; counter < array.length; counter++ )
31             System.out.printf( "%5d%8d\n", counter, array[ counter ] );
32       } // end else
33    } // end main
34 } // end class InitArray
```

```
java InitArray
Error: Please re-enter the entire command, including
an array size, initial value and increment.
```

```
java InitArray 5 0 4
Index   Value
    0       0
    1       4
    2       8
    3      12
    4      16
```

```
java InitArray 10 1 2
Index   Value
    0       1
    1       3
    2       5
    3       7
    4       9
    5      11
    6      13
    7      15
    8      17
    9      19
```

Overloaded Constructors

- Overloaded constructors
  - Provide multiple constructor definitions with different signatures
- No-argument constructor
  - A constructor invoked without arguments
- The `this` reference can be used to invoke another constructor
  - Allowed only as the first statement in a constructor's body

```java
// Fig. 8.5: Time2.java
// Time2 class declaration with overloaded constructors.

public class Time2
{
   private int hour;   // 0 - 23
   private int minute; // 0 - 59
   private int second; // 0 - 59

   // Time2 no-argument constructor: initializes each instance variable
   // to zero; ensures that Time2 objects start in a consistent state
   public Time2()
   {
      this( 0, 0, 0 ); // invoke Time2 constructor with three arguments
   } // end Time2 no-argument constructor

   // Time2 constructor: hour supplied, minute and second defaulted to 0
   public Time2( int h )
   {
      this( h, 0, 0 ); // invoke Time2 constructor with three arguments
   } // end Time2 one-argument constructor

   // Time2 constructor: hour and minute supplied, second defaulted to 0
   public Time2( int h, int m )
   {
      this( h, m, 0 ); // invoke Time2 constructor with three arguments
   } // end Time2 two-argument constructor

```

```
29    // Time2 constructor: hour, minute and second supplied
30    public Time2( int h, int m, int s )
31    {
32       setTime( h, m, s ); // invoke setTime to validate time
33    } // end Time2 three-argument constructor
34
35    // Time2 constructor: another Time2 object supplied
36    public Time2( Time2 time )
37    {
38       // invoke Time2 three-argument constructor
39       this( time.getHour(), time.getMinute(), time.getSecond() );
40    } // end Time2 constructor with a Time2 object argument
41
42    // Set Methods
43    // set a new time value using universal time; ensure that
44    // the data remains consistent by setting invalid values to zero
45    public void setTime( int h, int m, int s )
46    {
47       setHour( h );   // set the hour
48       setMinute( m ); // set the minute
49       setSecond( s ); // set the second
50    } // end method setTime
51
```

Garbage Collection and Method `finalize` (No pointers in Java)

- Garbage collection
  - JVM marks an object for garbage collection when there are no more references to that object
  - JVM's garbage collector will retrieve those objects memory so it can be used for other objects
- `finalize` method
  - All classes in Java have the `finalize` method
    - Inherited from the `Object` class
  - `finalize` is called by the garbage collector when it performs termination housekeeping
  - `finalize` takes no parameters and has return type `void`

```java
// Fig. 8.12: Employee.java
// Static variable used to maintain a count of the number of
// Employee objects in memory.

public class Employee
{
   private String firstName;
   private String lastName;
   private static int count = 0; // number of objects in memory

   // initialize employee, add 1 to static count and
   // output String indicating that constructor was called
   public Employee( String first, String last )
   {
      firstName = first;
      lastName = last;

      count++;  // increment static count of employees
      System.out.printf( "Employee constructor: %s %s; count = %d\n",
         firstName, lastName, count );
   } // end Employee constructor

```

```
23    // subtract 1 from static count when garbage
24    // collector calls finalize to clean up object;
25    // confirm that finalize was called
26    protected void finalize()
27    {
28       count--; // decrement static count of employees
29       System.out.printf( "Employee finalizer: %s %s; count = %d\n",
30          firstName, lastName, count );
31    } // end method finalize
32
33    // get first name
34    public String getFirstName()
35    {
36       return firstName;
37    } // end method getFirstName
38
39    // get last name
40    public String getLastName()
41    {
42       return lastName;
43    } // end method getLastName
44
45    // static method to get static count value
46    public static int getCount()
47    {
48       return count;
49    } // end method getCount
50 } // end class Employee
```

```java
// Fig. 8.13: EmployeeTest.java
// Static member demonstration.

public class EmployeeTest
{
   public static void main( String args[] )
   {
      // show that count is 0 before creating Employees
      System.out.printf( "Employees before instantiation: %d\n",
         Employee.getCount() );

      // create two Employees; count should be 2
      Employee e1 = new Employee( "Susan", "Baker" );
      Employee e2 = new Employee( "Bob", "Blue" );

```

```
16      // show that count is 2 after creating two Employees
17      System.out.println( "\nEmployees after instantiation: " );
18      System.out.printf( "via e1.getCount(): %d\n", e1.getCount() );
19      System.out.printf( "via e2.getCount(): %d\n", e2.getCount() );
20      System.out.printf( "via Employee.getCount(): %d\n",
21         Employee.getCount() );
22
23      // get names of Employees
24      System.out.printf( "\nEmployee 1: %s %s\nEmployee 2: %s %s\n\n",
25         e1.getFirstName(), e1.getLastName(),
26         e2.getFirstName(), e2.getLastName() );
27
28      // in this example, there is only one reference to each Employee,
29      // so the following two statements cause the JVM to mark each
30      // Employee object for garbage collection
31      e1 = null;
32      e2 = null;
33
34      System.gc(); // ask for garbage collection to occur now
35
```

```
36       // show Employee count after calling garbage collector; count
37       // displayed may be 0, 1 or 2 based on whether garbage collector
38       // executes immediately and number of Employee objects collected
39       System.out.printf( "\nEmployees after System.gc(): %d\n",
40          Employee.getCount() );
41    } // end main
42 } // end class EmployeeTest
```

```
Employees before instantiation: 0
Employee constructor: Susan Baker; count = 1
Employee constructor: Bob Blue; count = 2

Employees after instantiation:
via e1.getCount(): 2
via e2.getCount(): 2
via Employee.getCount(): 2

Employee 1: Susan Baker
Employee 2: Bob Blue

Employee finalizer: Bob Blue; count = 1
Employee finalizer: Susan Baker; count = 0

Employees after System.gc(): 0
```

`final` Instance Variables

- Principle of least privilege
  - Code should have only the privilege ad access it needs to accomplish its task, but no more
- `final` instance variables
  - Keyword `final`
    - Specifies that a variable is not modifiable (is a constant)
  - `final` instance variables can be initialized at their declaration
    - If they are not initialized in their declarations, they must be initialized in all constructors

```java
// Fig. 8.15: Increment.java
// final instance variable in a class.

public class Increment
{
   private int total = 0; // total of all increments
   private final int INCREMENT; // constant variable (uninitialized)

   // constructor initializes final instance variable INCREMENT
   public Increment( int incrementValue )
   {
      INCREMENT = incrementValue; // initialize constant variable (once)
   } // end Increment constructor

   // add INCREMENT to total
   public void addIncrementToTotal()
   {
      total += INCREMENT;
   } // end method addIncrementToTotal

   // return String representation of an Increment object's data
   public String toString()
   {
      return String.format( "total = %d", total );
   } // end method toIncrementString
} // end class Increment
```

```java
// Fig. 8.16: IncrementTest.java
// final variable initialized with a constructor argument.

public class IncrementTest
{
   public static void main( String args[] )
   {
      Increment value = new Increment( 5 );

      System.out.printf( "Before incrementing: %s\n\n", value );

      for ( int i = 1; i <= 3; i++ )
      {
         value.addIncrementToTotal();
         System.out.printf( "After increment %d: %s\n", i, value );
      } // end for
   } // end main
} // end class IncrementTest
```

```
Before incrementing: total = 0

After increment 1: total = 5
After increment 2: total = 10
After increment 3: total = 15
```

Creating Packages

- To declare a reusable class
  - Declare a `public` class
  - Add a `package` declaration to the source-code file
    - must be the very first executable statement in the file
    - `package` name should consist of your Internet domain name in reverse order followed by other names for the package
      - example: `com.deitel.jhtp6.ch08`
      - `package` name is part of the fully qualified class name
        - » Distinguishes between multiple classes with the same name belonging to different packages
        - » Prevents name conflict (also called name collision)
      - Class name without `package` name is the simple name

```java
// Fig. 8.18: Time1.java
// Time1 class declaration maintains the time in 24-hour format.
package com.deitel.jhtp6.ch08;

public class Time1
{
   private int hour;   // 0 - 23
   private int minute; // 0 - 59
   private int second; // 0 - 59

   // set a new time value using universal time; perform
   // validity checks on the data; set invalid values to zero
   public void setTime( int h, int m, int s )
   {
      hour = ( ( h >= 0 && h < 24 ) ? h : 0 );   // validate hour
      minute = ( ( m >= 0 && m < 60 ) ? m : 0 ); // validate minute
      second = ( ( s >= 0 && s < 60 ) ? s : 0 ); // validate second
   } // end method setTime

```

```
20    // convert to String in universal-time format (HH:MM:SS)
21    public String toUniversalString()
22    {
23       return String.format( "%02d:%02d:%02d", hour, minute, second );
24    } // end method toUniversalString
25
26    // convert to String in standard-time format (H:MM:SS AM or PM)
27    public String toString()
28    {
29       return String.format( "%d:%02d:%02d %s",
30          ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ),
31          minute, second, ( hour < 12 ? "AM" : "PM" ) );
32    } // end method toString
33 } // end class Time1
```

Inheritance

- Inheritance
  - Software reusability
  - Create new class from existing class
    - Absorb existing class's data and behaviors
    - Enhance with new capabilities
  - Subclass extends superclass
    - Subclass
      - More specialized group of objects
      - Behaviors inherited from superclass
        - » Can customize
      - Additional behaviors

```java
// Fig. 9.9: CommissionEmployee2.java
// CommissionEmployee2 class represents a commission employee.

public class CommissionEmployee2
{
   protected String firstName;
   protected String lastName;
   protected String socialSecurityNumber;
   protected double grossSales; // gross weekly sales
   protected double commissionRate; // commission percentage

   // five-argument constructor
   public CommissionEmployee2( String first, String last, String ssn,
      double sales, double rate )
   {
      // implicit call to Object constructor occurs here
      firstName = first;
      lastName = last;
      socialSecurityNumber = ssn;
      setGrossSales( sales ); // validate and store gross sales
      setCommissionRate( rate ); // validate and store commission rate
   } // end five-argument CommissionEmployee2 constructor

   // set first name
   public void setFirstName( String first )
   {
      firstName = first;
   } // end method setFirstName

```

```
30    // return first name
31    public String getFirstName()
32    {
33       return firstName;
34    } // end method getFirstName
35
36    // set last name
37    public void setLastName( String last )
38    {
39       lastName = last;
40    } // end method setLastName
41
42    // return last name
43    public String getLastName()
44    {
45       return lastName;
46    } // end method getLastName
47
48    // set social security number
49    public void setSocialSecurityNumber( String ssn )
50    {
51       socialSecurityNumber = ssn; // should validate
52    } // end method setSocialSecurityNumber
53
54    // return social security number
55    public String getSocialSecurityNumber()
56    {
57       return socialSecurityNumber;
58    } // end method getSocialSecurityNumber
59
```

```
60    // set gross sales amount
61    public void setGrossSales( double sales )
62    {
63       grossSales = ( sales < 0.0 ) ? 0.0 : sales;
64    } // end method setGrossSales
65
66    // return gross sales amount
67    public double getGrossSales()
68    {
69       return grossSales;
70    } // end method getGrossSales
71
72    // set commission rate
73    public void setCommissionRate( double rate )
74    {
75       commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
76    } // end method setCommissionRate
77
78    // return commission rate
79    public double getCommissionRate()
80    {
81       return commissionRate;
82    } // end method getCommissionRate
83
84    // calculate earnings
85    public double earnings()
86    {
87       return commissionRate * grossSales;
88    } // end method earnings
89
```

```
90    // return String representation of CommissionEmployee2 object
91    public String toString()
92    {
93       return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
94          "commission employee", firstName, lastName,
95          "social security number", socialSecurityNumber,
96          "gross sales", grossSales,
97          "commission rate", commissionRate );
98    } // end method toString
99 } // end class CommissionEmployee2
```

```java
// Fig. 9.10: BasePlusCommissionEmployee3.java
// BasePlusCommissionEmployee3 inherits from CommissionEmployee2 and has
// access to CommissionEmployee2's protected members.

public class BasePlusCommissionEmployee3 extends CommissionEmployee2
{
   private double baseSalary; // base salary per week

   // six-argument constructor
   public BasePlusCommissionEmployee3( String first, String last,
      String ssn, double sales, double rate, double salary )
   {
      super( first, last, ssn, sales, rate );
      setBaseSalary( salary ); // validate and store base salary
   } // end six-argument BasePlusCommissionEmployee3 constructor

   // set base salary
   public void setBaseSalary( double salary )
   {
      baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
   } // end method setBaseSalary

   // return base salary
   public double getBaseSalary()
   {
      return baseSalary;
   } // end method getBaseSalary

```

```
29    // calculate earnings
30    public double earnings()
31    {
32       return baseSalary + ( commissionRate * grossSales );
33    } // end method earnings
34
35    // return String representation of BasePlusCommissionEmployee3
36    public String toString()
37    {
38       return String.format(
39          "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
40          "base-salaried commission employee", firstName, lastName,
41          "social security number", socialSecurityNumber,
42          "gross sales", grossSales, "commission rate", commissionRate,
43          "base salary", baseSalary );
44    } // end method toString
45 } // end class BasePlusCommissionEmployee3
```

```java
// Fig. 9.11: BasePlusCommissionEmployeeTest3.java
// Testing class BasePlusCommissionEmployee3.

public class BasePlusCommissionEmployeeTest3
{
   public static void main( String args[] )
   {
      // instantiate BasePlusCommissionEmployee3 object
      BasePlusCommissionEmployee3 employee =
         new BasePlusCommissionEmployee3(
         "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );

      // get base-salaried commission employee data
      System.out.println(
         "Employee information obtained by get methods: \n" );
      System.out.printf( "%s %s\n", "First name is",
         employee.getFirstName() );
      System.out.printf( "%s %s\n", "Last name is",
         employee.getLastName() );
      System.out.printf( "%s %s\n", "Social security number is",
         employee.getSocialSecurityNumber() );
      System.out.printf( "%s %.2f\n", "Gross sales is",
         employee.getGrossSales() );
      System.out.printf( "%s %.2f\n", "Commission rate is",
         employee.getCommissionRate() );
      System.out.printf( "%s %.2f\n", "Base salary is",
         employee.getBaseSalary() );

```

```
29       employee.setBaseSalary( 1000 ); // set base salary
30
31       System.out.printf( "\n%s:\n\n%s\n",
32          "Updated employee information obtained by toString",
33          employee.toString() );
34    } // end main
35 } // end class BasePlusCommissionEmployeeTest3
```

```
Employee information obtained by get methods:

First name is Bob
Last name is Lewis
Social security number is 333-33-3333
Gross sales is 5000.00
Commission rate is 0.04
Base salary is 300.00

Updated employee information obtained by toString:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 1000.00
```

Controlling Access to Members of a Class

Access Levels				
Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

`Object` Class

- Class `Object` methods
  - `clone`
  - `equals`
  - `finalize`
  - `getClass`
  - `hashCode`
  - `notify`, `notifyAll`, `wait`
  - `toString`

Fig. 9.18 | `Object` methods that are inherited directly or indirectly by all classes. (Part 1 of 4)

Method	Description
`clone`	This `protected` method, which takes no arguments and returns an `Object` reference, makes a copy of the object on which it is called. When cloning is required for objects of a class, the class should override method `clone` as a `public` method and should implement interface `Cloneable` (package `java.lang`). The default implementation of this method performs a so-called shallow copy—instance variable values in one object are copied into another object of the same type. For reference types, only the references are copied. A typical overridden `clone` method's implementation would perform a deep copy that creates a new object for each reference type instance variable. There are many subtleties to overriding method `clone`. You can learn more about cloning in the following article: `java.sun.com/developer/JDCTechTips/2001/tt0306.html`

Fig. 9.18 | `Object` methods that are inherited directly or indirectly by all classes. (Part 2 of 4)

Method	Description
`Equals`	This method compares two objects for equality and returns `true` if they are equal and `false` otherwise. The method takes any `Object` as an argument. When objects of a particular class must be compared for equality, the class should override method `equals` to compare the contents of the two objects. The method's implementation should meet the following requirements: • It should return `false` if the argument is `null`. • It should return `true` if an object is compared to itself, as in `object1.equals( object1 )`. • It should return `true` only if both `object1.equals( object2 )` and `object2.equals( object1 )` would return `true`. • For three objects, if `object1.equals( object2 )` returns `true` and `object2.equals( object3 )` returns `true`, then `object1.equals( object3 )` should also return `true`. • If `equals` is called multiple times with the two objects and the objects do not change, the method should consistently return `true` if the objects are equal and `false` otherwise. A class that overrides `equals` should also override `hashCode` to ensure that equal objects have identical hashcodes. The default `equals` implementation uses operator `==` to determine whether two references *refer to the same object* in memory. Section 29.3.3 demonstrates class `String`'s `equals` method and differentiates between comparing `String` objects with `==` and with `equals`.

Fig. 9.18 | `Object` methods that are inherited directly or indirectly by all classes. (Part 3 of 4)

Method	Description
`finalize`	This `protected` method (introduced in Section 8.10 and Section 8.11) is called by the garbage collector to perform termination housekeeping on an object just before the garbage collector reclaims the object's memory. It is not guaranteed that the garbage collector will reclaim an object, so it cannot be guaranteed that the object's `finalize` method will execute. The method must specify an empty parameter list and must return `void`. The default implementation of this method serves as a placeholder that does nothing.
`getClass`	Every object in Java knows its own type at execution time. Method `getClass` (used in Section 10.5 and Section 21.3) returns an object of class `Class` (package `java.lang`) that contains information about the object's type, such as its class name (returned by `Class` method `getName`). You can learn more about class `Class` in the online API documentation at `java.sun.com/j2se/5.0/docs/api/java/lang/Class.html`.

Fig. 9.18 | `Object` methods that are inherited directly or indirectly by all classes. (Part 4 of 4)

Method	Description
`hashCode`	A hashtable is a data structure (discussed in Section 19.10) that relates one object, called the key, to another object, called the value. When initially inserting a value into a hashtable, the key's `hashCode` method is called. The hashcode value returned is used by the hashtable to determine the location at which to insert the corresponding value. The key's hashcode is also used by the hashtable to locate the key's corresponding value.
`notify`, `notifyAll`, `wait`	Methods `notify`, `notifyAll` and the three overloaded versions of `wait` are related to multithreading, which is discussed in Chapter 23. In J2SE 5.0, the multithreading model has changed substantially, but these features continue to be supported.
`toString`	This method (introduced in Section 9.4.1) returns a `String` representation of an object. The default implementation of this method returns the package name and class name of the object's class followed by a hexadecimal representation of the value returned by the object's `hashCode` method.

Polymorphism

- Polymorphism
  - When a program invokes a method through a superclass variable, the correct subclass version of the method is called, based on the type of the reference stored in the superclass variable
  - The same method name and signature can cause different actions to occur, depending on the type of object on which the method is invoked
  - Facilitates adding new classes to a system with minimal modifications to the system's code

Demonstrating Polymorphic Behavior

- A superclass reference can be aimed at a subclass object
  - This is possible because a subclass object *is a* superclass object as well
  - When invoking a method from that reference, the type of the actual referenced object, not the type of the reference, determines which method is called
- A subclass reference can be aimed at a superclass object only if the object is downcasted

```java
// Fig. 10.1: PolymorphismTest.java
// Assigning superclass and subclass references to superclass and
// subclass variables.

public class PolymorphismTest
{
   public static void main( String args[] )
   {
      // assign superclass reference to superclass variable
      CommissionEmployee3 commissionEmployee = new CommissionEmployee3(
         "Sue", "Jones", "222-22-2222", 10000, .06 );

      // assign subclass reference to subclass variable
      BasePlusCommissionEmployee4 basePlusCommissionEmployee =
         new BasePlusCommissionEmployee4(
         "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );

      // invoke toString on superclass object using superclass variable
      System.out.printf( "%s %s:\n\n%s\n\n",
         "Call CommissionEmployee3's toString with superclass reference ",
         "to superclass object", commissionEmployee.toString() );

      // invoke toString on subclass object using subclass variable
      System.out.printf( "%s %s:\n\n%s\n\n",
         "Call BasePlusCommissionEmployee4's toString with subclass",
         "reference to subclass object",
         basePlusCommissionEmployee.toString() );

```

```
29       // invoke toString on subclass object using superclass variable
30       CommissionEmployee3 commissionEmployee2 =
31          basePlusCommissionEmployee;
32       System.out.printf( "%s %s:\n\n%s\n",
33          "Call BasePlusCommissionEmployee4's toString with superclass",
34          "reference to subclass object", commissionEmployee2.toString() );
35    } // end main
36 } // end class PolymorphismTest
```

```
Call CommissionEmployee3's toString with superclass reference to superclass
object:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

Call BasePlusCommissionEmployee4's toString with subclass reference to
subclass object:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00

Call BasePlusCommissionEmployee4's toString with superclass reference to
subclass object:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00
```

Abstract Classes and Methods

- Abstract classes
  - Classes that are too general to create real objects
  - Used only as abstract superclasses for concrete subclasses and to declare reference variables
  - Many inheritance hierarchies have abstract superclasses occupying the top few levels
  - Keyword `abstract`
    - Use to declare a class `abstract`
    - Also use to declare a method `abstract`
      - Abstract classes normally contain one or more abstract methods
      - All concrete subclasses must override all inherited abstract methods

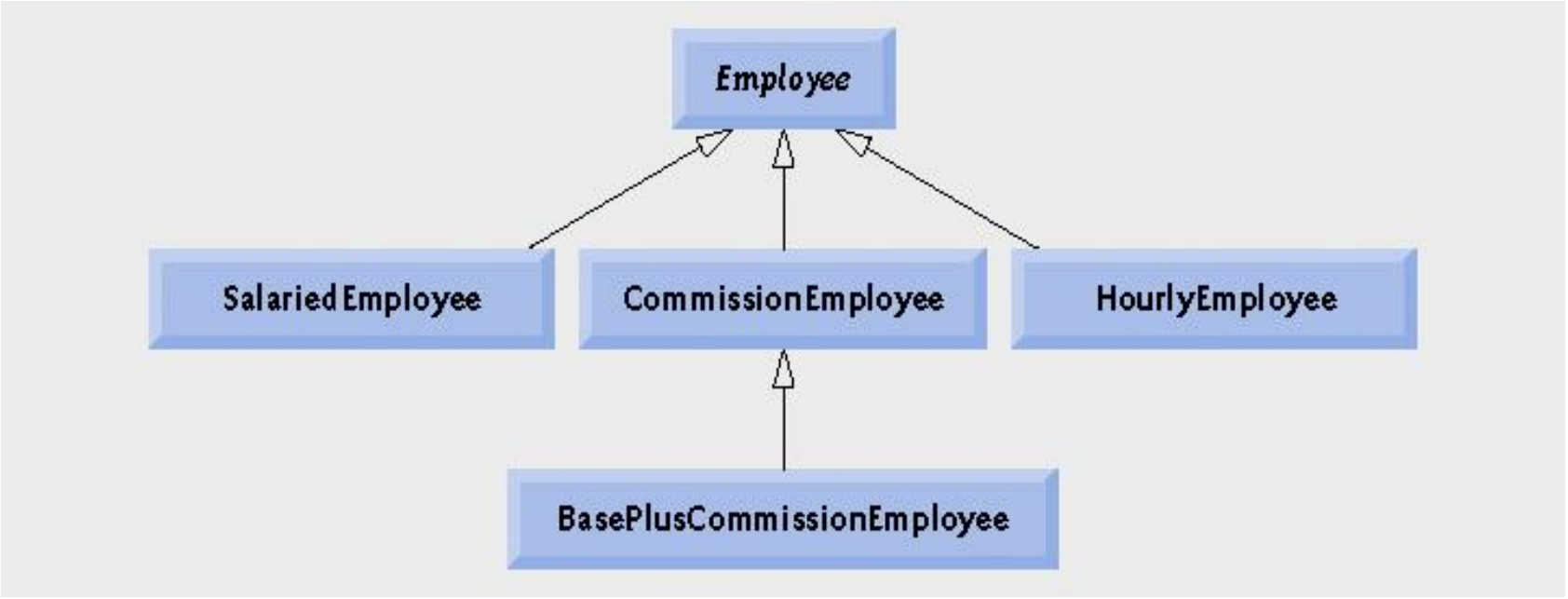
Abstract Classes and Methods

- `Iterator` class
  - Traverses all the objects in a collection, such as an array
  - Often used in polymorphic programming to traverse a collection that contains references to objects from various levels of a hierarchy

Creating Abstract Superclass `Employee`

- `abstract` superclass `Employee`
  - `earnings` is declared `abstract`
    - No implementation can be given for `earnings` in the `Employee` `abstract` class
  - An array of `Employee` variables will store references to subclass objects
    - `earnings` method calls from these variables will call the appropriate version of the `earnings` method

Employee hierarchy



Polymorphic interface for the `Employee` hierarchy classes.

	earnings	toString
Employee	abstract	*firstName lastName* social security number: *SSN*
Salaried-Employee	weeklySalary	salaried employee: *firstName lastName* social security number: *SSN* weekly salary: *weeklysalary*
Hourly-Employee	*If hours <= 40* wage * hours *If hours > 40* 40 * wage + (hours - 40) * wage * 1.5	hourly employee: *firstName lastName* social security number: *SSN* hourly wage: *wage*; hours worked: *hours*
Commission-Employee	commissionRate * grossSales	commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*
BasePlus-Commission-Employee	(commissionRate * grossSales) + baseSalary	base salaried commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*; base salary: *baseSalary*

```java
// Fig. 10.4: Employee.java
// Employee abstract superclass.

public abstract class Employee
{
   private String firstName;
   private String lastName;
   private String socialSecurityNumber;

   // three-argument constructor
   public Employee( String first, String last, String ssn )
   {
      firstName = first;
      lastName = last;
      socialSecurityNumber = ssn;
   } // end three-argument Employee constructor

```

```
18    // set first name
19    public void setFirstName( String first )
20    {
21       firstName = first;
22    } // end method setFirstName
23
24    // return first name
25    public String getFirstName()
26    {
27       return firstName;
28    } // end method getFirstName
29
30    // set last name
31    public void setLastName( String last )
32    {
33       lastName = last;
34    } // end method setLastName
35
36    // return last name
37    public String getLastName()
38    {
39       return lastName;
40    } // end method getLastName
41
```

```
42    // set social security number
43    public void setSocialSecurityNumber( String ssn )
44    {
45       socialSecurityNumber = ssn; // should validate
46    } // end method setSocialSecurityNumber
47
48    // return social security number
49    public String getSocialSecurityNumber()
50    {
51       return socialSecurityNumber;
52    } // end method getSocialSecurityNumber
53
54    // return String representation of Employee object
55    public String toString()
56    {
57       return String.format( "%s %s\nsocial security number: %s",
58          getFirstName(), getLastName(), getSocialSecurityNumber() );
59    } // end method toString
60
61    // abstract method overridden by subclasses
62    public abstract double earnings(); // no implementation here
63 } // end abstract class Employee
```

```java
// Fig. 10.5: SalariedEmployee.java
// SalariedEmployee class extends Employee.

public class SalariedEmployee extends Employee
{
   private double weeklySalary;

   // four-argument constructor
   public SalariedEmployee( String first, String last, String ssn,
      double salary )
   {
      super( first, last, ssn ); // pass to Employee constructor
      setWeeklySalary( salary ); // validate and store salary
   } // end four-argument SalariedEmployee constructor

   // set salary
   public void setWeeklySalary( double salary )
   {
      weeklySalary = salary < 0.0 ? 0.0 : salary;
   } // end method setWeeklySalary

```

```
22    // return salary
23    public double getWeeklySalary()
24    {
25       return weeklySalary;
26    } // end method getWeeklySalary
27
28    // calculate earnings; override abstract method earnings in Employee
29    public double earnings()
30    {
31       return getWeeklySalary();
32    } // end method earnings
33
34    // return String representation of SalariedEmployee object
35    public String toString()
36    {
37       return String.format( "salaried employee: %s\n%s: $%,.2f",
38          super.toString(), "weekly salary", getWeeklySalary() );
39    } // end method toString
40 } // end class SalariedEmployee
```

```java
// Fig. 10.6: HourlyEmployee.java
// HourlyEmployee class extends Employee.

public class HourlyEmployee extends Employee
{
   private double wage; // wage per hour
   private double hours; // hours worked for week

   // five-argument constructor
   public HourlyEmployee( String first, String last, String ssn,
      double hourlyWage, double hoursWorked )
   {
      super( first, last, ssn );
      setWage( hourlyWage ); // validate hourly wage
      setHours( hoursWorked ); // validate hours worked
   } // end five-argument HourlyEmployee constructor

   // set wage
   public void setWage( double hourlyWage )
   {
      wage = ( hourlyWage < 0.0 ) ? 0.0 : hourlyWage;
   } // end method setWage

   // return wage
   public double getWage()
   {
      return wage;
   } // end method getWage

```

```
30    // set hours worked
31    public void setHours( double hoursWorked )
32    {
33       hours = ( ( hoursWorked >= 0.0 ) && ( hoursWorked <= 168.0 ) ) ?
34          hoursWorked : 0.0;
35    } // end method setHours
36
37    // return hours worked
38    public double getHours()
39    {
40       return hours;
41    } // end method getHours
42
43    // calculate earnings; override abstract method earnings in Employee
44    public double earnings()
45    {
46       if ( getHours() <= 40 ) // no overtime
47          return getWage() * getHours();
48       else
49          return 40 * getWage() + ( gethours() - 40 ) * getWage() * 1.5;
50    } // end method earnings
51
52    // return String representation of HourlyEmployee object
53    public String toString()
54    {
55       return String.format( "hourly employee: %s\n%s: $%,.2f; %s: %,.2f",
56          super.toString(), "hourly wage", getWage(),
57          "hours worked", getHours() );
58    } // end method toString
59 } // end class HourlyEmployee
```

```java
// Fig. 10.7: CommissionEmployee.java
// CommissionEmployee class extends Employee.

public class CommissionEmployee extends Employee
{
   private double grossSales; // gross weekly sales
   private double commissionRate; // commission percentage

   // five-argument constructor
   public CommissionEmployee( String first, String last, String ssn,
      double sales, double rate )
   {
      super( first, last, ssn );
      setGrossSales( sales );
      setCommissionRate( rate );
   } // end five-argument CommissionEmployee constructor

   // set commission rate
   public void setCommissionRate( double rate )
   {
      commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
   } // end method setCommissionRate

```

```
24    // return commission rate
25    public double getCommissionRate()
26    {
27       return commissionRate;
28    } // end method getCommissionRate
29
30    // set gross sales amount
31    public void setGrossSales( double sales )
32    {
33       grossSales = ( sales < 0.0 ) ? 0.0 : sales;
34    } // end method setGrossSales
35
36    // return gross sales amount
37    public double getGrossSales()
38    {
39       return grossSales;
40    } // end method getGrossSales
41
```

```
42    // calculate earnings; override abstract method earnings in Employee
43    public double earnings()
44    {
45       return getCommissionRate() * getGrossSales();
46    } // end method earnings
47
48    // return String representation of CommissionEmployee object
49    public String toString()
50    {
51       return String.format( "%s: %s\n%s: $%,.2f; %s: %.2f",
52          "commission employee", super.toString(),
53          "gross sales", getGrossSales(),
54          "commission rate", getCommissionRate() );
55    } // end method toString
56 } // end class CommissionEmployee
```

```java
// Fig. 10.8: BasePlusCommissionEmployee.java
// BasePlusCommissionEmployee class extends CommissionEmployee.

public class BasePlusCommissionEmployee extends CommissionEmployee
{
   private double baseSalary; // base salary per week

   // six-argument constructor
   public BasePlusCommissionEmployee( String first, String last,
      String ssn, double sales, double rate, double salary )
   {
      super( first, last, ssn, sales, rate );
      setBaseSalary( salary ); // validate and store base salary
   } // end six-argument BasePlusCommissionEmployee constructor

   // set base salary
   public void setBaseSalary( double salary )
   {
      baseSalary = ( salary < 0.0 ) ? 0.0 : salary; // non-negative
   } // end method setBaseSalary

```

```
22    // return base salary
23    public double getBaseSalary()
24    {
25       return baseSalary;
26    } // end method getBaseSalary
27
28    // calculate earnings; override method earnings in CommissionEmployee
29    public double earnings()
30    {
31       return getBaseSalary() + super.earnings();
32    } // end method earnings
33
34    // return String representation of BasePlusCommissionEmployee object
35    public String toString()
36    {
37       return String.format( "%s %s; %s: $%,.2f",
38          "base-salaried", super.toString(),
39          "base salary", getBaseSalary() );
40    } // end method toString
41 } // end class BasePlusCommissionEmployee
```

```java
// Fig. 10.9: PayrollSystemTest.java
// Employee hierarchy test program.

public class PayrollSystemTest
{
   public static void main( String args[] )
   {
      // create subclass objects
      SalariedEmployee salariedEmployee =
         new SalariedEmployee( "John", "Smith", "111-11-1111", 800.00 );
      HourlyEmployee hourlyEmployee =
         new HourlyEmployee( "Karen", "Price", "222-22-2222", 16.75, 40 );
      CommissionEmployee commissionEmployee =
         new CommissionEmployee(
         "Sue", "Jones", "333-33-3333", 10000, .06 );
      BasePlusCommissionEmployee basePlusCommissionEmployee =
         new BasePlusCommissionEmployee(
         "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );

      System.out.println( "Employees processed individually:\n" );

```

```
22        System.out.printf( "%s\n%s: $%,.2f\n\n",
23           salariedEmployee, "earned", salariedEmployee.earnings() );
24        System.out.printf( "%s\n%s: $%,.2f\n\n",
25           hourlyEmployee, "earned", hourlyEmployee.earnings() );
26        System.out.printf( "%s\n%s: $%,.2f\n\n",
27           commissionEmployee, "earned", commissionEmployee.earnings() );
28        System.out.printf( "%s\n%s: $%,.2f\n\n",
29           basePlusCommissionEmployee,
30           "earned", basePlusCommissionEmployee.earnings() );
31
32        // create four-element Employee array
33        Employee employees[] = new Employee[ 4 ];
34
35        // initialize array with Employees
36        employees[ 0 ] = salariedEmployee;
37        employees[ 1 ] = hourlyEmployee;
38        employees[ 2 ] = commissionEmployee;
39        employees[ 3 ] = basePlusCommissionEmployee;
40
41        System.out.println( "Employees processed polymorphically:\n" );
42
43        // generically process each element in array employees
44        for ( Employee currentEmployee : employees )
45        {
46           System.out.println( currentEmployee ); // invokes toString
47
```

```
48          // determine whether element is a BasePlusCommissionEmployee
49          if ( currentEmployee instanceof BasePlusCommissionEmployee )
50          {
51             // downcast Employee reference to
52             // BasePlusCommissionEmployee reference
53             BasePlusCommissionEmployee employee =
54                ( BasePlusCommissionEmployee ) currentEmployee;
55
56             double oldBaseSalary = employee.getBaseSalary();
57             employee.setBaseSalary( 1.10 * oldBaseSalary );
58             System.out.printf(
59                "new base salary with 10%% increase is: $%,.2f\n",
60                employee.getBaseSalary() );
61          } // end if
62
63          System.out.printf(
64             "earned $%,.2f\n\n", currentEmployee.earnings() );
65       } // end for
66
67       // get type name of each object in employees array
68       for ( int j = 0; j < employees.length; j++ )
69          System.out.printf( "Employee %d is a %s\n", j,
70             employees[ j ].getClass().getName() );
71    } // end main
72 } // end class PayrollSystemTest
```

```
Employees processed individually:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: $800.00
earned: $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: $16.75; hours worked: 40.00
earned: $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: $10,000.00; commission rate: 0.06
earned: $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: $5,000.00; commission rate: 0.04; base salary: $300.00
earned: $500.00
```

```
Employees processed polymorphically:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: $800.00
earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: $16.75; hours worked: 40.00
earned $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: $10,000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: $5,000.00; commission rate: 0.04; base salary: $300.00
new base salary with 10% increase is: $330.00
earned $530.00

Employee 0 is a SalariedEmployee
Employee 1 is a HourlyEmployee
Employee 2 is a CommissionEmployee
Employee 3 is a BasePlusCommissionEmployee
```

Operator `instanceof` and Downcasting

- Dynamic binding
  - Also known as late binding
  - Calls to overridden methods are resolved at execution time, based on the type of object referenced
- `instanceof` operator
  - Determines whether an object is an instance of a certain type

Operator `instanceof` and Downcasting (Cont.)

- Downcasting
  - Convert a reference to a superclass to a reference to a subclass
  - Allowed only if the object has an *is-a* relationship with the subclass
- `getClass` method
  - Inherited from `Object`
  - Returns an object of type `Class`
- `getName` method of class `Class`
  - Returns the class's name

`final` Methods and Classes

- `final` methods
  - Cannot be overridden in a subclass
  - `private` and `static` methods are implicitly `final`
  - `final` methods are resolved at compile time, this is known as static binding
    - Compilers can optimize by inlining the code
- `final` classes
  - Cannot be extended by a subclass
  - All methods in a `final` class are implicitly `final`

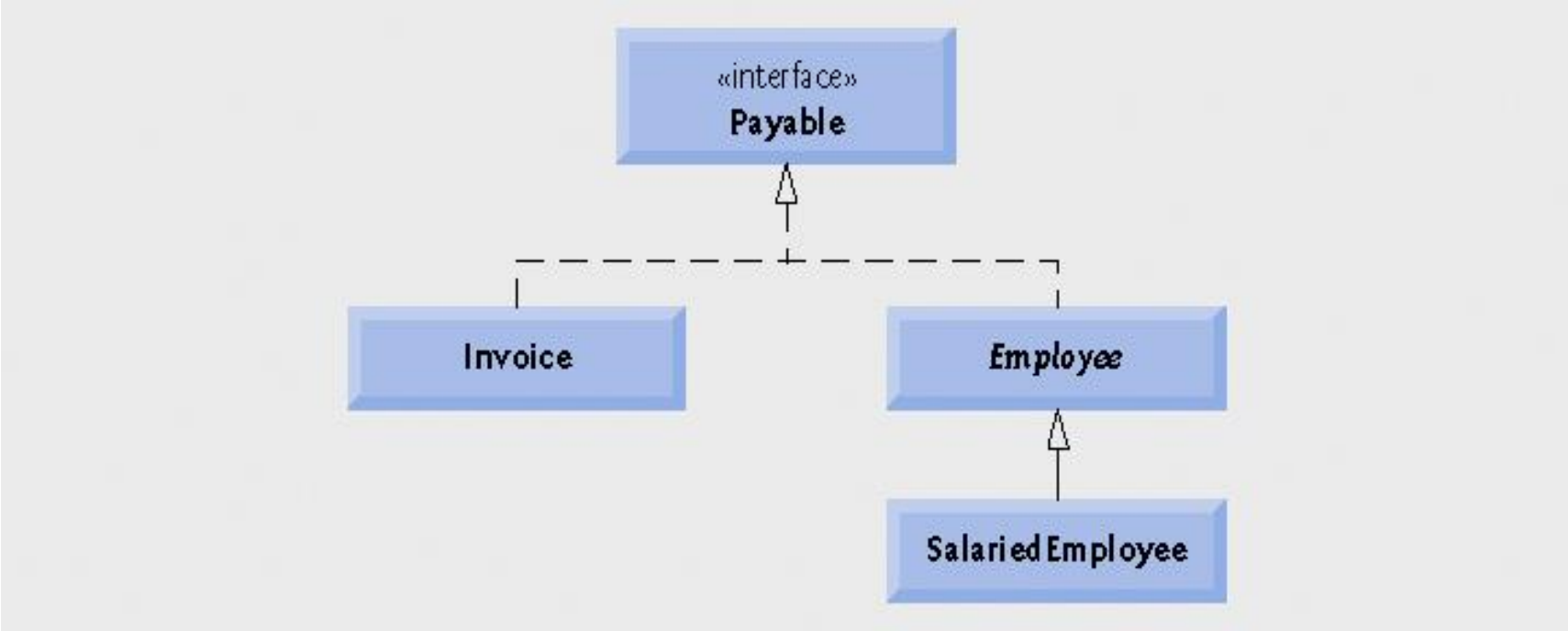
Creating and Using Interfaces

- Interfaces
  - Keyword `interface`
  - Contains only constants and `abstract` methods
    - All fields are implicitly `public`, `static` and `final`
    - All methods are implicitly `public abstract` methods
  - Classes can `implement` interfaces
    - The class must declare each method in the interface using the same signature or the class must be declared `abstract`
  - Typically used when disparate classes need to share common methods and constants
  - Normally declared in their own files with the same names as the interfaces and with the `.java` file-name extension

Developing a `Payable` Hierarchy

- `Payable` interface
  - Contains method `getPaymentAmount`
  - Is implemented by the `Invoice` and `Employee` classes
- The relationship between a class and an interface is known as realization
  - A class “realizes” the method of an interface

Payable interface hierarchy



```java
// Fig. 10.11: Payable.java
// Payable interface declaration.

public interface Payable
{
   double getPaymentAmount(); // calculate payment; no implementation
} // end interface Payable
```

Creating Class `Invoice`

- A class can implement as many interfaces as it needs
  - Use a comma-separated list of interface names after keyword implements
    - Example: `public class` *ClassName* `extends` *SuperclassName* `implements` *FirstInterface*`,` *SecondInterface*`, ...`

```java
// Fig. 10.12: Invoice.java
// Invoice class implements Payable.

public class Invoice implements Payable
{
   private String partNumber;
   private String partDescription;
   private int quantity;
   private double pricePerItem;

   // four-argument constructor
   public Invoice( String part, String description, int count,
      double price )
   {
      partNumber = part;
      partDescription = description;
      setQuantity( count ); // validate and store quantity
      setPricePerItem( price ); // validate and store price per item
   } // end four-argument Invoice constructor

   // set part number
   public void setPartNumber( String part )
   {
      partNumber = part;
   } // end method setPartNumber

```

```
27    // get part number
28    public String getPartNumber()
29    {
30       return partNumber;
31    } // end method getPartNumber
32
33    // set description
34    public void setPartDescription( String description )
35    {
36       partDescription = description;
37    } // end method setPartDescription
38
39    // get description
40    public String getPartDescription()
41    {
42       return partDescription;
43    } // end method getPartDescription
44
45    // set quantity
46    public void setQuantity( int count )
47    {
48       quantity = ( count < 0 ) ? 0 : count; // quantity cannot be negative
49    } // end method setQuantity
50
51    // get quantity
52    public int getQuantity()
53    {
54       return quantity;
55    } // end method getQuantity
56
```

```
57    // set price per item
58    public void setPricePerItem( double price )
59    {
60       pricePerItem = ( price < 0.0 ) ? 0.0 : price; // validate price
61    } // end method setPricePerItem
62
63    // get price per item
64    public double getPricePerItem()
65    {
66       return pricePerItem;
67    } // end method getPricePerItem
68
69    // return String representation of Invoice object
70    public String toString()
71    {
72       return String.format( "%s: \n%s: %s (%s) \n%s: %d \n%s: $%,.2f",
73          "invoice", "part number", getPartNumber(), getPartDescription(),
74          "quantity", getQuantity(), "price per item", getPricePerItem() );
75    } // end method toString
76
77    // method required to carry out contract with interface Payable
78    public double getPaymentAmount()
79    {
80       return getQuantity() * getPricePerItem(); // calculate total cost
81    } // end method getPaymentAmount
82 } // end class Invoice
```

```java
// Fig. 10.13: Employee.java
// Employee abstract superclass implements Payable.

public abstract class Employee implements Payable
{
   private String firstName;
   private String lastName;
   private String socialSecurityNumber;

   // three-argument constructor
   public Employee( String first, String last, String ssn )
   {
      firstName = first;
      lastName = last;
      socialSecurityNumber = ssn;
   } // end three-argument Employee constructor

```

```
18    // set first name
19    public void setFirstName( String first )
20    {
21       firstName = first;
22    } // end method setFirstName
23
24    // return first name
25    public String getFirstName()
26    {
27       return firstName;
28    } // end method getFirstName
29
30    // set last name
31    public void setLastName( String last )
32    {
33       lastName = last;
34    } // end method setLastName
35
36    // return last name
37    public String getLastName()
38    {
39       return lastName;
40    } // end method getLastName
41
```

```
42    // set social security number
43    public void setSocialSecurityNumber( String ssn )
44    {
45       socialSecurityNumber = ssn; // should validate
46    } // end method setSocialSecurityNumber
47
48    // return social security number
49    public String getSocialSecurityNumber()
50    {
51       return socialSecurityNumber;
52    } // end method getSocialSecurityNumber
53
54    // return String representation of Employee object
55    public String toString()
56    {
57       return String.format( "%s %s\nsocial security number: %s",
58          getFirstName(), getLastName(), getSocialSecurityNumber() );
59    } // end method toString
60
61    // Note: We do not implement Payable method getPaymentAmount here so
62    // this class must be declared abstract to avoid a compilation error.
63 } // end abstract class Employee
```

Modifying Class `SalariedEmployee` for Use in the `Payable` Hierarchy

- Objects of any subclasses of the class that implements the interface can also be thought of as objects of the interface
  - A reference to a subclass object can be assigned to an interface variable if the superclass implements that interface

```java
// Fig. 10.14: SalariedEmployee.java
// SalariedEmployee class extends Employee, which implements Payable.

public class SalariedEmployee extends Employee
{
   private double weeklySalary;

   // four-argument constructor
   public SalariedEmployee( String first, String last, String ssn,
      double salary )
   {
      super( first, last, ssn ); // pass to Employee constructor
      setWeeklySalary( salary ); // validate and store salary
   } // end four-argument SalariedEmployee constructor

   // set salary
   public void setWeeklySalary( double salary )
   {
      weeklySalary = salary < 0.0 ? 0.0 : salary;
   } // end method setWeeklySalary

```

```
22    // return salary
23    public double getWeeklySalary()
24    {
25       return weeklySalary;
26    } // end method getWeeklySalary
27
28    // calculate earnings; implement interface Payable method that was
29    // abstract in superclass Employee
30    public double getPaymentAmount()
31    {
32       return getWeeklySalary();
33    } // end method getPaymentAmount
34
35    // return String representation of SalariedEmployee object
36    public String toString()
37    {
38       return String.format( "salaried employee: %s\n%s: $%,.2f",
39          super.toString(), "weekly salary", getWeeklySalary() );
40    } // end method toString
41 } // end class SalariedEmployee
```

```java
// Fig. 10.15: PayableInterfaceTest.java
// Tests interface Payable.

public class PayableInterfaceTest
{
   public static void main( String args[] )
   {
      // create four-element Payable array
      Payable payableObjects[] = new Payable[ 4 ];

      // populate array with objects that implement Payable
      payableObjects[ 0 ] = new Invoice( "01234", "seat", 2, 375.00 );
      payableObjects[ 1 ] = new Invoice( "56789", "tire", 4, 79.95 );
      payableObjects[ 2 ] =
         new SalariedEmployee( "John", "Smith", "111-11-1111", 800.00 );
      payableObjects[ 3 ] =
         new SalariedEmployee( "Lisa", "Barnes", "888-88-8888", 1200.00 );

      System.out.println(
         "Invoices and Employees processed polymorphically:\n" );

```

```
22       // generically process each element in array payableObjects
23       for ( Payable currentPayable : payableObjects )
24       {
25          // output currentPayable and its appropriate payment amount
26          System.out.printf( "%s \n%s: $%,.2f\n\n",
27             currentPayable.toString(),
28             "payment due", currentPayable.getPaymentAmount() );
29       } // end for
30    } // end main
31 } // end class PayableInterfaceTest
```

```
Invoices and Employees processed polymorphically:

invoice:
part number: 01234 (seat)
quantity: 2
price per item: $375.00
payment due: $750.00

invoice:
part number: 56789 (tire)
quantity: 4
price per item: $79.95
payment due: $319.80

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: $800.00
payment due: $800.00

salaried employee: Lisa Barnes
social security number: 888-88-8888
weekly salary: $1,200.00
payment due: $1,200.00
```

Fig. 10.16 | Common interfaces of the Java API. (Part 1 of 2)

Interface	Description
`Comparable`	As you learned in Chapter 2, Java contains several comparison operators (e.g., <, <=, >, >=, ==, !=) that allow you to compare primitive values. However, these operators cannot be used to compare the contents of objects. Interface `Comparable` is used to allow objects of a class that implements the interface to be compared to one another. The interface contains one method, `compareTo`, that compares the object that calls the method to the object passed as an argument to the method. Classes must implement `compareTo` such that it returns a value indicating whether the object on which it is invoked is less than (negative integer return value), equal to (`0` return value) or greater than (positive integer return value) the object passed as an argument, using any criteria specified by the programmer. For example, if class `Employee` implements `Comparable`, its `compareTo` method could compare `Employee` objects by their earnings amounts. Interface `Comparable` is commonly used for ordering objects in a collection such as an array. We use `Comparable` in Chapter 18, Generics, and Chapter 19, Collections.
`Serializable`	A tagging interface used only to identify classes whose objects can be written to (i.e., serialized) or read from (i.e., deserialized) some type of storage (e.g., file on disk, database field) or transmitted across a network. We use `Serializable` in Chapter 14, Files and Streams, and Chapter 24, Networking.

Fig. 10.16 | Common interfaces of the Java API. (Part 2 of 2)

Interface	Description
`Runnable`	Implemented by any class for which objects of that class should be able to execute in parallel using a technique called multithreading (discussed in Chapter 23, Multithreading). The interface contains one method, `run`, which describes the behavior of an object when executed.
GUI event-listener interfaces	You work with Graphical User Interfaces (GUIs) every day. For example, in your Web browser, you might type in a text field the address of a Web site to visit, or you might click a button to return to the previous site you visited. When you type a Web site address or click a button in the Web browser, the browser must respond to your interaction and perform the desired task for you. Your interaction is known as an event, and the code that the browser uses to respond to an event is known as an event handler. In Chapter 11, GUI Components: Part 1, and Chapter 22, GUI Components: Part 2, you will learn how to build Java GUIs and how to build event handlers to respond to user interactions. The event handlers are declared in classes that implement an appropriate event-listener interface. Each event listener interface specifies one or more methods that must be implemented to respond to user interactions.
`SwingConstants`	Contains a set of constants used in GUI programming to position GUI elements on the screen. We explore GUI programming in Chapters 11 and 22.

Cloning objects

slides created by Marty Stepp
based on materials by M. Ernst, S. Reges, D. Notkin, R. Mercer, Wikipedia
http://www.cs.washington.edu/331/

Copying objects

- In other languages (common in C++), to enable clients to easily make copies of an object, you can supply a *copy constructor* :

```
// in client code
Point p1 = new Point(-3, 5);
Point p2 = new Point(p1);      // make p2 a copy of p1


// in Point.java
public Point(Point blueprint) {   // copy constructor
    this.x = blueprint.x;
    this.y = blueprint.y;
}
```

  - Java has some copy constructors but also has a different way...

Object clone method

```
protected Object clone()
        throws CloneNotSupportedException
```

- Creates and returns a copy of this object. General intent:
  - `x.clone() != x`
  - `x.clone().equals(x)`
  - `x.clone().getClass() == x.getClass()`
    - (though none of the above are absolute requirements)

- The `Object` class's `clone` method makes a "shallow copy" of the object, but by convention, the object returned by this method should be **independent** of this object (which is being cloned).

Protected access

```
protected Object clone()
        throws CloneNotSupportedException
```

- **protected**: Visible only to the class itself, its subclasses, and any other classes in the same package.
  - In other words, for most classes you are not allowed to call `clone`.
  - If you want to enable cloning, you must override `clone`.
    - You should make it `public` so clients can call it.
    - You can also change the return type to your class's type. (good)
    - You can also not throw the exception. (good)
  - You must also make your class implement the `Cloneable` interface to signify that it is allowed to be cloned.

The Cloneable interface

```
public interface Cloneable {}
```

- Why would there ever be an interface with no methods?
  - Another example: `Set` interface, a sub-interface of `Collection`

- **tagging interface**: One that does not contain/add any methods, but is meant to mark a class as having a certain quality or ability.
  - Generally a wart in the Java language; a misuse of interfaces.
  - Now largely unnecessary thanks to *annotations* (seen later).
  - But we still must interact with a few tagging interfaces, like this one.

- Let's implement clone for a `Point` class...

Flawed clone method 1

```
public class Point implements Cloneable {
    private int x, y;
    ...
    public Point clone() {
        Point copy = new Point(this.x, this.y);
        return copy;
    }
}
```

- What's wrong with the above method?

The flaw

```
// also implements Cloneable and inherits clone()
public class Point3D extends Point {
    private int z;
    ...
}
```

- The above `Point3D` class's `clone` method produces a `Point`!
  - This is undesirable and unexpected behavior.
  - The only way to ensure that the clone will have exactly the same type as the original object (even in the presence of inheritance) is to call the `clone` method from class `Object` with `super.clone()`.

Proper clone method

```
public class Point implements Cloneable {
    private int x, y;
    ...
    public Point clone() {
        try {
            Point copy = (Point) super.clone();
            return copy;
        } catch (CloneNotSupportedException e) {
            // this will never happen
            return null;
        }
    }
}
```

- To call `Object`'s `clone` method, you must use `try/catch`.
  - But if you implement `Cloneable`, the exception will not be thrown.

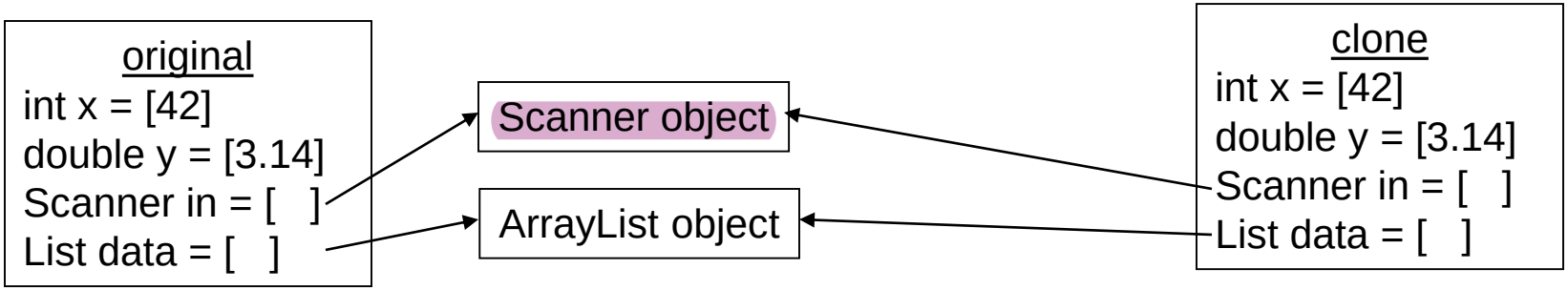
Flawed clone method 2

```
public class BankAccount implements Cloneable {
    private String name;
    private List<String> transactions;
    ...
    public BankAccount clone() {
        try {
            BankAccount copy = (BankAccount) super.clone();
            return copy;
        } catch (CloneNotSupportedException e) {
            return null;   // won't ever happen
        }
    }
}
```

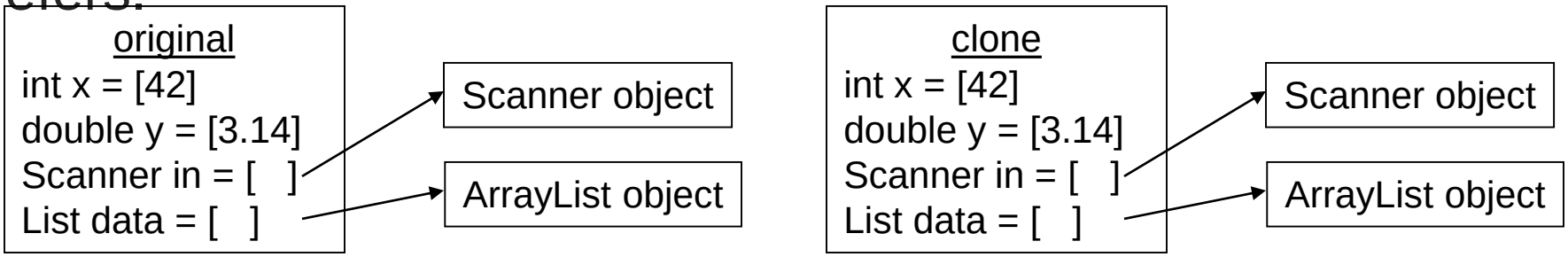
- What's wrong with the above method?

Shallow vs. deep copy

- **shallow copy**: Duplicates an object without duplicating any other objects to which it refers.



- **deep copy**: Duplicates an object's entire *reference graph*: copies itself and deep copies any other objects to which it refers.



  - `Object`'s `clone` method makes a shallow copy by default. (Why?)

Proper clone method 2

```
public class BankAccount implements Cloneable {
   private String name;
   private List<String> transactions;
   ...
   public BankAccount clone() {
      try {               // deep copy
         BankAccount copy = (BankAccount) super.clone();
         copy.transactions = new ArrayList<String>(transactions);
         return copy;
      } catch (CloneNotSupportedException e) {
         return null;   // won't ever happen
      }
   }
}
```

– Copying the list of transactions (and any other modifiable reference fields) produces a deep copy that is independent of the original.

Effective Java Tip #11

- **Tip #11**: Override `clone` judiciously.

- Cloning has many gotchas and warts:
  - protected vs. public
  - flaws in the presence of inheritance
  - requires the use of an ugly tagging interface
  - throws an ugly checked exception
  - easy to get wrong by making a shallow copy instead of a deep copy

Java Exceptions

- Exception – an indication of a problem that occurs during a program's execution
- Exception handling – resolving exceptions that may occur so program can continue or terminate gracefully
- Exception handling enables programmers to create programs that are more robust and fault-tolerant

Using the `throws` Clause

- `throws` clause – specifies the exceptions a method may throw
  - Appears after method's parameter list and before the method's body
  - Contains a comma-separated list of exceptions
  - Exceptions can be thrown by statements in method's body of by methods called in method's body
  - Exceptions can be of types listed in `throws` clause or subclasses

```java
// Fig. 13.2: DivideByZeroWithExceptionHandling.java
// An exception-handling example that checks for divide-by-zero.
import java.util.InputMismatchException;
import java.util.Scanner;

public class DivideByZeroWithExceptionHandling
{
   // demonstrates throwing an exception when a divide-by-zero occurs
   public static int quotient( int numerator, int denominator )
      throws ArithmeticException
   {
      return numerator / denominator; // possible division by zero
   } // end method quotient

   public static void main( String args[] )
   {
      Scanner scanner = new Scanner( System.in ); // scanner for input
      boolean continueLoop = true; // determines if more input is needed

      do
      {
         try // read two numbers and calculate quotient
         {
            System.out.print( "Please enter an integer numerator: " );
            int numerator = scanner.nextInt();
            System.out.print( "Please enter an integer denominator: " );
            int denominator = scanner.nextInt();

```

```
29             int result = quotient( numerator, denominator );
30             System.out.printf( "\nResult: %d / %d = %d\n", numerator,
31                denominator, result );
32             continueLoop = false; // input successful; end looping
33          } // end try
34          catch ( InputMismatchException inputMismatchException )
35          {
36             System.err.printf( "\nException: %s\n",
37                inputMismatchException );
38             scanner.nextLine(); // discard input so user can try again
39             System.out.println(
40                "You must enter integers. Please try again.\n" );
41          } // end catch
42          catch ( ArithmeticException arithmeticException )
43          {
44             System.err.printf( "\nException: %s\n", arithmeticException );
45             System.out.println(
46                "Zero is an invalid denominator. Please try again.\n" );
47          } // end catch
48       } while ( continueLoop ); // end do...while
49    } // end main
50 } // end class DivideByZeroWithExceptionHandling
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 0

Exception: java.lang.ArithmeticException: / by zero
Zero is an invalid denominator. Please try again.

Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: hello

Exception: java.util.InputMismatchException
You must enter integers. Please try again.

Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

Throwing Exceptions Using the `throw` Statement

- `throw` statement – used to throw exceptions
- Programmers can thrown exceptions themselves from a method if something has gone wrong
- `throw` statement consists of keyword `throw` followed by the exception object

```java
// Fig. 13.5: UsingExceptions.java
// Demonstration of the try...catch...finally exception handling
// mechanism.

public class UsingExceptions
{
   public static void main( String args[] )
   {
      try
      {
         throwException(); // call method throwException
      } // end try
      catch ( Exception exception ) // exception thrown by throwException
      {
         System.err.println( "Exception handled in main" );
      } // end catch

      doesNotThrowException();
   } // end main

```

```
21    // demonstrate try...catch...finally
22    public static void throwException() throws Exception
23    {
24       try // throw an exception and immediately catch it
25       {
26          System.out.println( "Method throwException" );
27          throw new Exception(); // generate exception
28       } // end try
29       catch ( Exception exception ) // catch exception thrown in try
30       {
31          System.err.println(
32             "Exception handled in method throwException" );
33          throw exception; // rethrow for further processing
34
35          // any code here would not be reached
36
37       } // end catch
38       finally // executes regardless of what occurs in try...catch
39       {
40          System.err.println( "Finally executed in throwException" );
41       } // end finally
42
43       // any code here would not be reached, exception rethrown in catch
44
```

```
45    } // end method throwException
46
47    // demonstrate finally when no exception occurs
48    public static void doesNotThrowException()
49    {
50       try // try block does not throw an exception
51       {
52          System.out.println( "Method doesNotThrowException" );
53       } // end try
54       catch ( Exception exception ) // does not execute
55       {
56          System.err.println( exception );
57       } // end catch
58       finally // executes regardless of what occurs in try...catch
59       {
60          System.err.println(
61             "Finally executed in doesNotThrowException" );
62       } // end finally
63
64       System.out.println( "End of method doesNotThrowException" );
65    } // end method doesNotThrowException
66 } // end class UsingExceptions
```

```
Method throwException
Exception handled in method throwException
Finally executed in throwException
Exception handled in main
Method doesNotThrowException
Finally executed in doesNotThrowException
End of method doesNotThrowException
```

Generics

- Generics
  - New feature of J2SE 5.0
  - Provide compile-time type safety
    - Catch invalid types at compile time
  - Generic methods
    - A single method declaration
    - A set of related methods
  - Generic classes
    - A single class declaration
    - A set of related clases

```java
// Fig. 18.3: GenericMethodTest.java
// Using generic methods to print array of different types.

public class GenericMethodTest
{
   // generic method printArray
   public static < E > void printArray( E[] inputArray )
   {
      // display array elements
      for ( E element : inputArray )
         System.out.printf( "%s ", element );

      System.out.println();
   } // end method printArray

   public static void main( String args[] )
   {
      // create arrays of Integer, Double and Character
      Integer[] intArray = { 1, 2, 3, 4, 5 };
      Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
      Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

```

```
23         System.out.println( "Array integerArray contains:" );
24         printArray( integerArray ); // pass an Integer array
25         System.out.println( "\nArray doubleArray contains:" );
26         printArray( doubleArray ); // pass a Double array
27         System.out.println( "\nArray characterArray contains:" );
28         printArray( characterArray ); // pass a Character array
29      } // end main
30   } // end class GenericMethodTest
```

```
Array integerArray contains:
1 2 3 4 5 6

Array doubleArray contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7

Array characterArray contains:
H E L L O
```

Javadoc comments

```
/**
 *  description of class/method/field/etc.
 *
 *  @tag  attributes
 *  @tag  attributes
 *  ...
 *  @tag  attributes
 */
```

- **Javadoc comments**: Special comment syntax for describing detailed specifications of Java classes and methods.
  - Put on all class headers, public methods, constructors, public fields, ...
    - *Main benefit:* Tools can turn Javadoc comments into HTML spec pages.
  - Eclipse and other editors have useful built-in Javadoc support.
    - *Main drawback:* Comments can become bulky and harder to read.

Javadoc tags

- on a method or constructor:

tag	description
`@param` ***name description***	describes a parameter
`@return` ***description***	describes what value will be returned
`@throws` ***ExceptionType reason***	describes an exception that may be thrown (and what would cause it to be thrown)
`{@code` ***sourcecode*** `}`	for showing Java code in the comments
`{@inheritDoc}`	allows a subclass method to copy Javadoc comments from the superclass version

- on a class header:

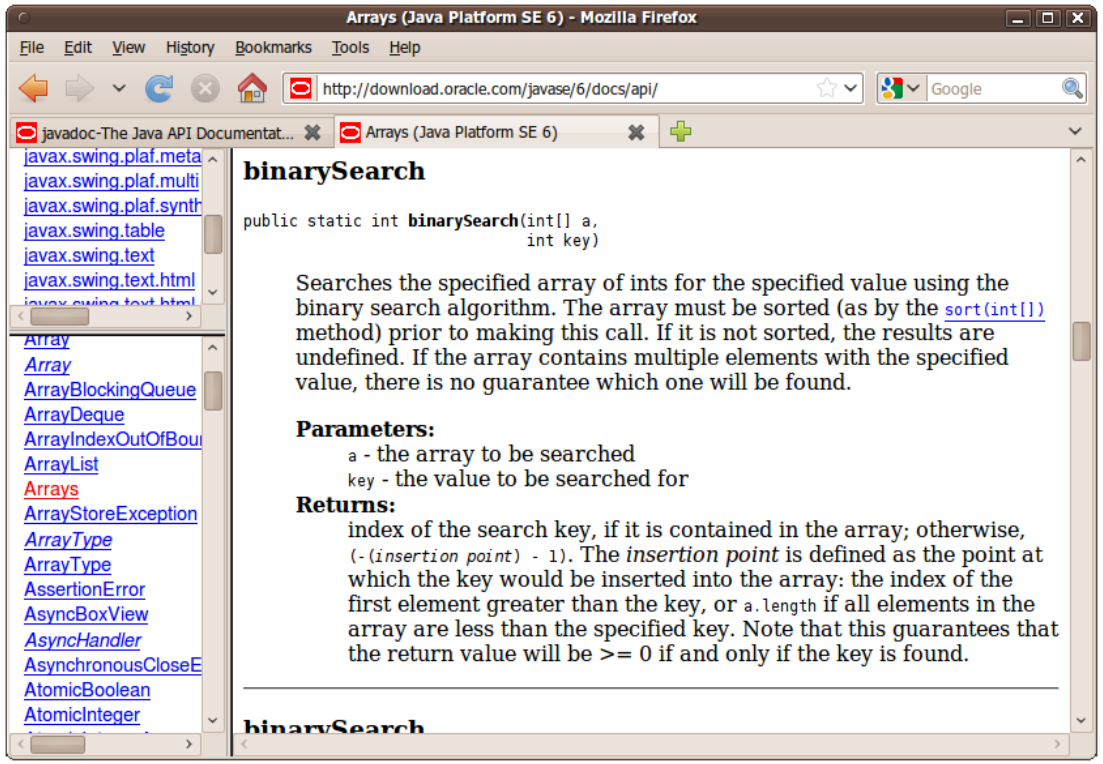
tag	description
`@author` ***name***	author of a class
`@version` ***number***	class's version number, in any format

Javadoc example

```
/**
 * Each BankAccount object models the account information for
 * a single user of Fells Wargo bank.
 * @author James T. Kirk
 * @version 1.4 (Aug 9 2008)
 */
public class BankAccount {
    /** The standard interest rate on all accounts. */
    public static final double INTEREST_RATE = 0.03;
    ...

    /**
     * Deducts the given amount of money from this account's
     * balance, if possible, and returns whether the money was
     * deducted successfully (true if so, false if not).
     * If the account does not contain sufficient funds to
     * make this withdrawal, no funds are withdrawn.
     *
     * @param amount the amount of money to be withdrawn
     * @return true if amount was withdrawn, else false
     * @throws IllegalArgumentException if amount is negative
     */
    public boolean withdraw(double amount) {
        ...
    }
}
```

Javadoc output as HTML

- Java includes tools to convert Javadoc comments into web pages
  - from Terminal: `javadoc -d doc/ *.java`
  - Eclipse has this built in: Project → Generate Javadoc...
  - The actual Java API spec web pages are generated from Sun's Javadoc comments on their own source code:



Javadoc HTML example

- from `java.util.List` interface source code:

```
/**
 * Returns the element at the specified position
 * in this list.
 * <p>This method is <em>not</em> guaranteed to run
 * in constant time. In some implementations it may
 * run in time proportional to the element position.
 *
 * @param index index of element to return; must be
 *         non-negative and less than size of this list
 * @return the element at the specified position
 * @throws IndexOutOfBoundsException if the index is
 *         out of range
 *         ({@code index < 0 || index >= this.size()})
 */
public E get(int index);
```

  - Notice that HTML tags may be embedded inside the comments.

Javadoc enums, constants

- Each class constant or enumeration value can be commented:

```
/**
 * An instrument section of a symphony orchestra.
 * @author John Williams
 */
public enum OrchestraSection {
    /** Woodwinds, such as flute, clarinet, and oboe. */
    WOODWIND,

    /** Brass instruments, such as trumpet. */
    BRASS,

    /** Percussion instruments, such as cymbals. */
    PERCUSSION,

    /** Stringed instruments, such as violin and cello. */
    STRING;
}
```

What goes in @param/return

- Don't repeat yourself or write vacuous comments.

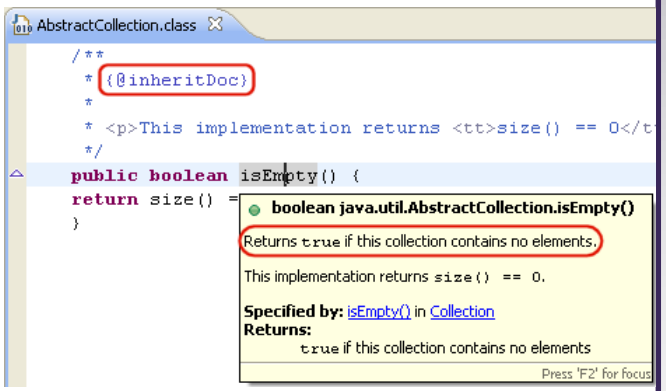
```
/** Takes an index and element and adds the element there.
 * @param index index to use
 * @param element element to add
 */
public boolean add(int index, E element) { ...
```

- better:

```
/** Inserts the specified element at the specified
 * position in this list.  Shifts the element currently at
 * that position (if any) and any subsequent elements to
 * the right (adds one to their indices).  Returns whether
 * the add was successful.
 * @param index index at which the element is to be inserted
 * @param element element to be inserted at the given index
 * @return true if added successfully; false if not
 * @throws IndexOutOfBoundsException if index out of range
 *         ({@code index < 0 || index > size()})
 */
public boolean add(int index, E element) { ...
```

Your Javadoc is your spec

- **Whenever you write a class to be used by clients, you should write full Javadoc comments for all of its public behavior.**
  - This constitutes your specification to all clients for your class.
  - You can post the generated HTML files publicly for clients to view.
  - Common distribution of a library of classes:
    - **binaries** (.class files, often packaged into an archive)
    - **specification** (Javadoc .html files, or a public URL to view them)
  - Eclipse uses Javadoc for auto-completion.



- **Effective Java Tip #44:**
  Write Javadoc comments for all exposed API elements.

  (anything that is non-private)

Javadoc and private

- Private internal methods do not need Javadoc comments:

```
/** ... a Javadoc comment ... */
public void remove(int index) { ... }

// Helper does the real work of removing
// the item at the given index.
private void removeHelper(int index) {
    for (int i = index; i < size - 1; i++) {
        elementData[i] = elementData[i + 1];
    }
    elementData[size - 1] = 0;
    size--;
}
```

  - Private members do not appear in the generated HTML pages.

Custom Javadoc tags

- Javadoc doesn't have tags for pre/post, but you can add them:

tag	description
`@pre` ***`condition`*** *`(or @precondition)`*	notes a precondition in API documentation; describes a condition that must be true for the method to perform its functionality
`@post` ***`condition`*** *`(or @postcondition)`*	notes a postcondition in API documentation; describes a condition that is guaranteed to be true at the *end* of the method's functionality, so long as all preconditions were true at the *start* of the method

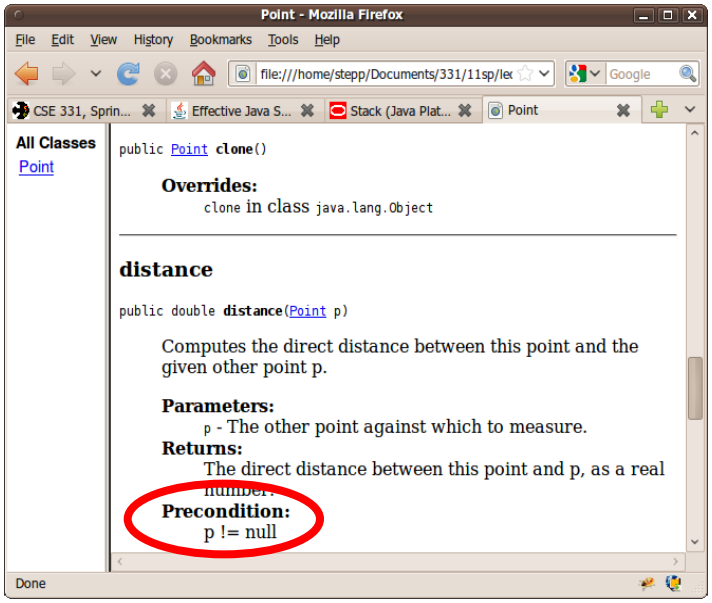
  - By default, these tags won't show up in the generated HTML. But...

Applying custom Javadoc tags

- from Terminal: `javadoc -d doc/`
  `-tag pre:cm:"Precondition:"`
  `-tag post:cm:"Postcondition:" *.java`
- in Eclipse: Project → Generate Javadoc... → Next → Next →
  in the "Extra Javadoc options" box, type:

```
-tag pre:cm:"Precondition:" -tag post:cm:"Postcondition:"
```

- The generated Java API web pages will now be able to display `pre` and `post` tags properly!



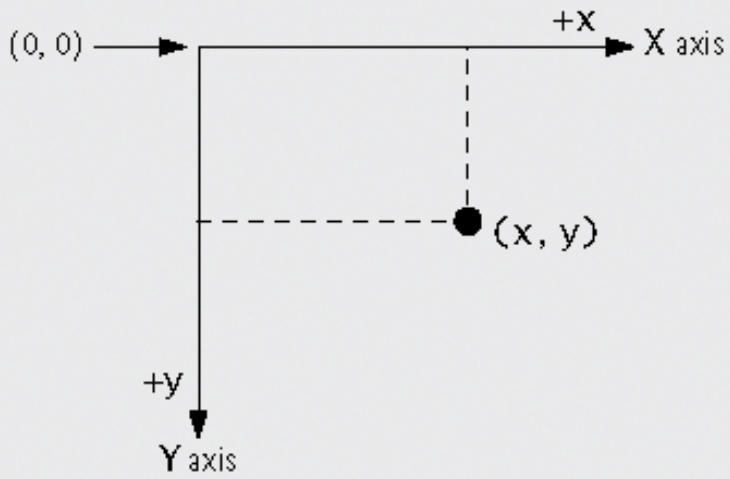
Some basic GUI components.

Component	Description
`JLabel`	**Displays uneditable text or icons.**
`JTextField`	**Enables user to enter data from the keyboard. Can also be used to display editable or uneditable text.**
`JButton`	**Triggers an event when clicked with the mouse.**
`JCheckBox`	**Specifies an option that can be selected or not selected.**
`JComboBox`	**Provides a drop-down list of items from which the user can make a selection by clicking an item or possibly by typing into the box.**
`JList`	**Provides a list of items from which the user can make a selection by clicking on any item in the list. Multiple elements can be selected.**
`JPanel`	**Provides an area in which components can be placed and organized. Can also be used as a drawing area for graphics.**

GUI and Graphics Case Study: Creating Simple Drawings

- Java's coordinate system
  - Defined by x-coordinates and y-coordinates
    - Also known as horizontal and vertical coordinates
    - Are measured along the x-axis and y-axis
  - Coordinate units are measured in pixels
- `Graphics` class from the `java.awt` package
  - Provides methods for drawing text and shapes
- `JPanel` class from the `javax.swing` package
  - Provides an area on which to draw

Java coordinate system. Units are measured in pixels.



```java
// Fig. 4.19: DrawPanel.java
// Draws two crossing lines on a panel.
import java.awt.Graphics;
import javax.swing.JPanel;

public class DrawPanel extends JPanel
{
   // draws an X from the corners of the panel
   public void paintComponent( Graphics g )
   {
      // call paintComponent to ensure the panel displays correctly
      super.paintComponent( g );

      int width = getWidth(); // total width
      int height = getHeight(); // total height

      // draw a line from the upper-left to the lower-right
      g.drawLine( 0, 0, width, height );

      // draw a line from the lower-left to the upper-right
      g.drawLine( 0, height, width, 0 );
   } // end method paintComponent
} // end class DrawPanel
```

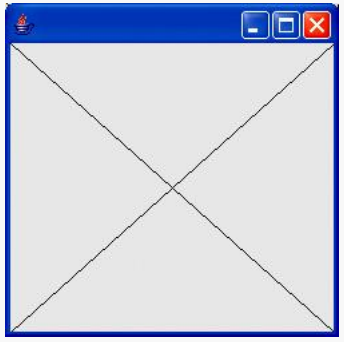
GUI and Graphics Case Study: Creating Simple Drawings (

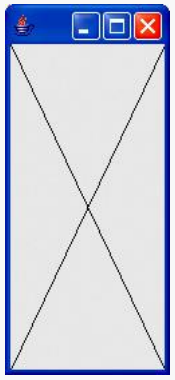
- The `JPanel` class
  - Every `JPanel` has a `paintComponent` method
    - `paintComponent` is called whenever the system needs to display the `JPanel`
  - `getWidth` and `getHeight` methods
    - Return the width and height of the `JPanel`, respectively
  - `drawLine` method
    - Draws a line from the coordinates defined by its first two arguments to the coordinates defined by its second two arguments

GUI and Graphics Case Study: Creating Simple Drawings

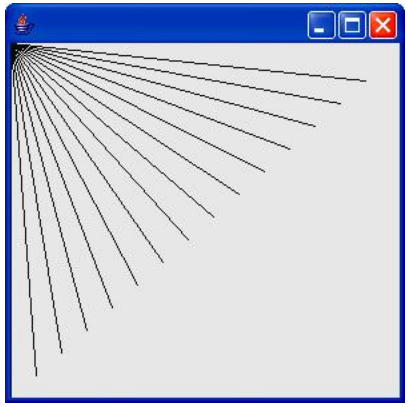
- `JFrame` class from the `javax.swing` package
  - Allows the programmer to create a window
  - `setDefaultCloseOperation` method
    - Pass `JFrame.EXIT_ON_CLOSE` as its argument to set the application to terminate when the user closes the window
  - `add` method
    - Attaches a `JPanel` to the `JFrame`
  - `setSize` method
    - Sets the width (first argument) and height (second argument) of the `JFrame`

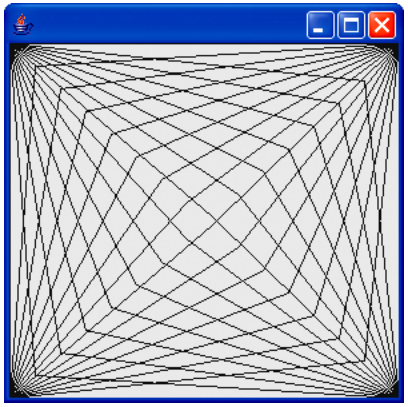
```java
// Fig. 4.20: DrawPanelTest.java
// Application to display a DrawPanel.
import javax.swing.JFrame;

public class DrawPanelTest
{
   public static void main( String args[] )
   {
      // create a panel that contains our drawing
      DrawPanel panel = new DrawPanel();

      // create a new frame to hold the panel
      JFrame application = new JFrame();

      // set the frame to exit when it is closed
      application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

      application.add( panel ); // add the panel to the frame
      application.setSize( 250, 250 ); // set the size of the frame
      application.setVisible( true ); // make the frame visible
   } // end main
} // end class DrawPanelTest
```



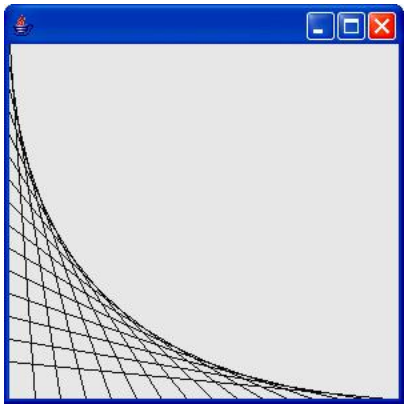


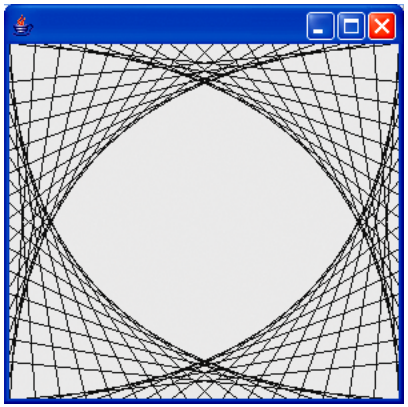
Lines fanning from a corner.





Line art with loops and drawLine.





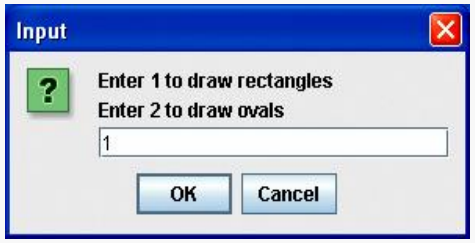
GUI and Graphics Case Study: Drawing Rectangles and Ovals

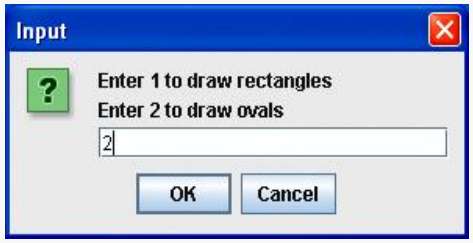
- Draw rectangles
  - Method `drawRect` of `Graphics`
- Draw ovals
  - Method `drawOval` of `Graphics`

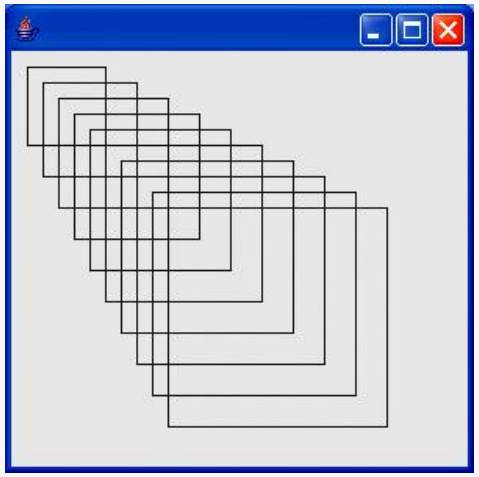
```java
// Fig. 5.26: Shapes.java
// Demonstrates drawing different shapes.
import java.awt.Graphics;
import javax.swing.JPanel;

public class Shapes extends JPanel
{
   private int choice; // user's choice of which shape to draw

   // constructor sets the user's choice
   public Shapes( int userChoice )
   {
      choice = userChoice;
   } // end Shapes constructor

   // draws a cascade of shapes starting from the top left corner
   public void paintComponent( Graphics g )
   {
      super.paintComponent( g );

```

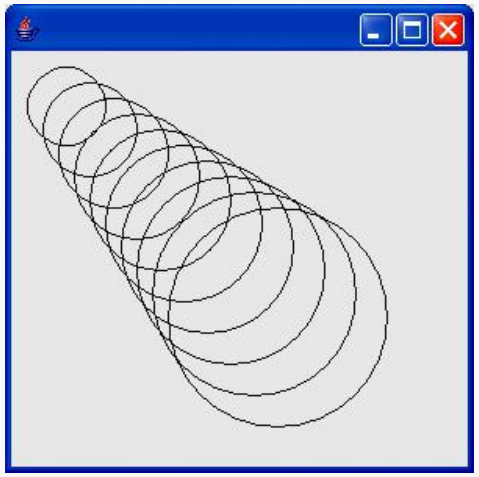
```
21       for ( int i = 0; i < 10; i++ )
22       {
23          // pick the shape based on the user's choice
24          switch ( choice )
25          {
26             case 1: // draw rectangles
27                g.drawRect( 10 + i * 10, 10 + i * 10,
28                   50 + i * 10, 50 + i * 10 );
29                break;
30             case 2: // draw ovals
31                g.drawOval( 10 + i * 10, 10 + i * 10,
32                   50 + i * 10, 50 + i * 10 );
33                break;
34          } // end switch
35       } // end for
36    } // end method paintComponent
37 } // end class Shapes
```

```java
// Fig. 5.27: ShapesTest.java
// Test application that displays class Shapes.
import javax.swing.JFrame;
import javax.swing.JOptionPane;

public class ShapesTest
{
   public static void main( String args[] )
   {
      // obtain user's choice
      String input = JOptionPane.showInputDialog(
         "Enter 1 to draw rectangles\n" +
         "Enter 2 to draw ovals" );

      int choice = Integer.parseInt( input ); // convert input to int

      // create the panel with the user's input
      Shapes panel = new Shapes( choice );

      JFrame application = new JFrame(); // creates a new JFrame

      application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      application.add( panel ); // add the panel to the frame
      application.setSize( 300, 300 ); // set the desired size
      application.setVisible( true ); // show the frame
   } // end main
} // end class ShapesTest
```


Input
Enter 1 to draw rectangles
Enter 2 to draw ovals
1
OK
Cancel


Input
Enter 1 to draw rectangles
Enter 2 to draw ovals
2
OK
Cancel





Drawing concentric circles.