Part 1: Introduction

Course Contents

- Difference between conventional and object-oriented programming
- Introduction to object-oriented programming with Java
- … lots of Java details …
- … aimed at producing and maintaining medium sized software systems.

Goals

- Sound knowledge of programming principles
- Sound knowledge of object-orientation
- Able to critically assess the quality of small and medium software systems
- Able to implement a medium-sized software systems in Java

Books

- Java How to program
  Deitel & Deitel, Prentice Hall

- Objects First with Java A Practical Introduction using BlueJ

- Object-Oriented Software Construction
  Bertrand Meyer (Pearson 1997), ISBN 0136291554
Sources for Slides (Acknowledgements)

- David J. Barnes & Michael Kölling
- Sigi Benkner, University of Vienna, Inst. f. Software Science
- Donald Bell, California State Polytechnic University, Pomona
- Deitel & Deitel, Prentice Hall
- T. Fahringer, Inst. of Computer Science, UIBK

Webpage

The course web page is at
dps.uibk.ac.at/~tf/lehre/ss04/seII/Vorlesung.html

There is also a mailing list for this course. Please check them regularly. They will be used for announcements and distribution of material.

Course overview (1)

- Introduction to Computer
- Introduction to Java Applications
- Introduction to Java Applets
- Control Structures
- Arrays
- Object Oriented Programming
- Inheritance
- Polymorphism
- Exception Handling
- and much more…

Software Engineering

- Software Engineering should help to reach similar quality levels for large software systems as achieved for classical engineering disciplines.

- Goals:
  - cost-effective software development
  - high quality
  - meet all time constraints
**Example: Erroneous Software**

- 22. Juli 1962, Cape Canaveral/Florida:
  Start of the 1st US Venus satellite Mariner 1.
- Excerpt of a Fortran program to control the carrier rocket Atlas-Agena B (NASA):

```fortran
IF (TVAL .LT. 0.2E-2) GOTO 40
DO 40 M = 1, 3
  W0 = (M-1) *0.5
  DO 5 K = 1, 3
    T(K) = W0
    Z = 1.0/(X**2) *B1**2+3.09977E-4 *BO**2
    D(K) = 3.076E-2*2*(1.0/X*BO*B1+3.0977480**2*X**2*BO**2)-1)/Z
    E(K) = H**2*93.2943*WO/SIN (WO)*Z
    H = D(K) - E(K)
    5 CONTINUE
  10 CONTINUE
  Y = H/WO-1
40 CONTINUE
```

**Definition: „Software Engineering“**

- F.L. Bauer (NATO Konferenz, 1968)
  „The establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines."
- Standard Glossar IEEE (1983)
  „... Is the systematic approach to the development, operation, maintenance, and retirement of software."

**Main Task of Software Development**

- Development
- Quality Assurance
- Management
- Maintenance

**Software**

- The term software comprises a set of artifacts:
  - programs
  - documents
  - data
- Software products are developed for a certain customer (custom software) or for the general market (generic software).
Costs and Complexity of Software

<table>
<thead>
<tr>
<th>Products</th>
<th>COST (US million)</th>
<th>EFFORT (Man Year)</th>
<th>LINES OF CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lotus 1-2-3 V.3</td>
<td>22</td>
<td>263</td>
<td>400,000</td>
</tr>
<tr>
<td>Space Shuttle</td>
<td>1200</td>
<td>22096</td>
<td>25,600,000</td>
</tr>
<tr>
<td>CitiBank Teller</td>
<td>13</td>
<td>150</td>
<td>780,000</td>
</tr>
</tbody>
</table>

A paramount problem of software development is the handling of complexity.

Trend towards more and more complex software:
- Distributed applications (Internet, Grid, Mobile Computing,...)
- Multimedia
- Real time applications

Traditional Software Development

- Waterfall model
  - Dominated software development for a long time
  - Widely applied through many different variations

- Main problems:
  - Strict classification of software projects
  - Discrete transition among phases
  - Late discovery of design errors (*late design breakage*)
  - Late appearance of actual code
  - Risk not to meet requirements
  - Impedance mismatches

Traditional Software Development (1)

- Structured Analysis and Design (SA/SD)
  - Basis are analysis techniques developed between 1965 u. 1975.
  - Center: Modeling of processes through functions.
  - Widely applied (conventional software development)
  - Main focus: Implementation and Test
  - Complex maintenance
Traditional Software Development (2)

- Software Errors during Design and Implementation
  - 75% of errors found before release
  - Coding: 64%
  - Analyse/Design: 36%
  - 30% of errors found before release

Costs of Error Correction

- Definition: 1x
- Development: 1.5-6x
- After release: 60-100x

Object-Oriented Software Development

At the end of the 80s a change of paradigm in software engineering occurred: object orientation

- Object-oriented Analysis and Design (OOA/OOD)
  - Based on OO technology (starting in 1985)
  - Object models are central (union of data and process models)
  - Centric: analysis and design
  - Simplified maintenance

Object-oriented versus Procedural

- Object-oriented Paradigm
  - Data and Operations (Methods) define unit → Class
  - Center issue: What
  - Smalltalk, C++, Java, ...

- In contrast: procedural programming
  - Separation of data and methods (procedures)
  - Center issue: How
  - Pascal, C, Fortran, ...
**OOA/OOD versus SA/SD**

- **OOA/OOD**
  - Classification based on concepts (objects)
- **SA/SD**
  - Classification based on processes (functions)

---

**What are Objects?**

- To a child they are something perceptible by one or more of the senses.

- To an IT professional they are a software packets that abstract important features of a real object.

---

**Introduction to Objects**

- What are Objects?
- The Three Keys to Object Technology
- Definitions

---

**What are Objects?**

- Real objects are such things as:
  - Porsche
What are Objects?

- Real objects are such things as:
  - Los Angeles
  - Alan Kay (Customer)
  - Versateller
  - Things Porsche
  - Places Los Angeles
  - Roles Alan Kay
  - Systems Versateller
What are Objects?

• Real objects have characteristics:
  1. Porsches  Top Speed
  2. Los Angeles  Population
  3. Customers  Balanced Owed
  4. Versatellers  Amount on Hand

• Real objects also have behaviors:
  1. Porsches  Accelerate
  2. Los Angeles  Tax
  3. Customers  Purchase
  4. Versatellers  Dispense Cash

• Objekts have characteristics and behavior

• The Three Keys to Object Technology

  Key 1  A software object has both characteristics (variables) and behavior (methods) encapsulated in it.
  Key 2  Software objects communicate by messages.
  Key 3  Software objects can inherit characteristics and behavior just like many real objects.
The Three Keys to Object Technology

• Key one: Characteristics are encapsulated by the objects behavior

• Key two: Objects communicate by messages

  – triangle.redraw()
  – ellipse.redraw()
  – rectangle.redraw()

Polymorphism means different implementations can be hidden behind a common interface.

• Key three: Objects can be classified into classes

  – Classes are a template that is used to manufacture objects (instances)
  (Note that instance is a synonym of object)
The Three Keys to Object Technology

- **Key three (also):**
  Classes contain the behavior called methods while the instances contain the characteristics called variables.

### Definitions

- **Object**
  A software packet that abstracts the salient behavior and characteristics of a real object into a software package that simulates the real object.
Definitions

• Encapsulation
The containment of the data behind a software membrane consisting of methods. The data can only be accessed through the encapsulated behavior.

• Message
A signal from a client object requesting services from a server object.
The message may contain arguments.
The server object may return a response.

• Class
A template for defining new object instances.
It is sometimes called an object class.
The methods reside in the object class.

• Instance
A term used to refer to an actual object.
Sometimes called an object instance.
Definitions

• Inheritance

A technique to allow classes to use a parent classes methods and data.

Inheritance can have many levels

• Polymorphism

A request handling mechanism that selects a method depending on the message and the target object.

Thus the message means different things to different objects.

• Overriding

Where an overloaded method is chosen from the child class instead of from the parent class.

• Method

The actual computer code that is encapsulated in the object class.
### Definitions

- **Subclass**

  A class that has a parent class to inherit from.

### Summary

- Object technology is the current modern paradigm
- OO can enable reuse of system design and coding effort (and consequently save a lot of time and money)
- Systems built with object technology should be able to be built faster, be more reliable, and be more maintainable
- Use of object technology requires a different way of thinking than does traditional programming
Part 2:
Introduction to Computers

1.2 What Is a Computer?

- Computer
  - Performs computations and makes logical decisions
  - Millions / billions times faster than human beings
- Computer programs
  - Sets of instructions for which computer processes data
- Hardware
  - Physical devices of computer system
- Software
  - Programs that run on computers
- Operating System
  - Controls resources (CPU, memory, disc, network)
  - Distributes resources and uses resources as well
  - Interface for programmers

1.3 Computer Organization (1)

- Six logical units of a computer system
  - Input unit
    - Mouse, keyboard
  - Output unit
  - Memory unit
    - Retains input and processed information
  - Arithmetic and logic unit (ALU)
    - Performs calculations
  - Central processing unit (CPU)
    - Supervises operation of other devices
  - Secondary storage unit
    - Hard drives, floppy drives

Computer Organization (2)

- Computers have many devices
  - Input/Output Devices
  - memory
  - Processor(s)

Building blocks of a simple PC
1.4 Evolution of Operating Systems

- Batch processing
  - One job (task) at a time
  - Operating systems developed
    - Programs to make computers more convenient to use
    - Switch jobs easier
- Multiprogrammed batch systems
  - Multiple jobs share CPU without user interaction
- Time-sharing
  - Timesharing operating systems
  - User interaction possible

1.5 Personal, Distributed and Client/Server Computing

- Personal computing
  - Computers for personal use
- Distributed computing
  - Computing performed among several computers
- Client/server computing
  - Servers offer common store of programs and data
  - Clients access programs and data from server
1.6 Machine Languages, Assembly Languages and High-Level Languages

- **Machine language**
  - “Natural language” of computer component
  - Machine dependent

- **Assembly language**
  - English-like abbreviations represent computer operations
  - Translator (assembler) converts assembly to machine language

- **High-level language**
  - Allows for writing more “English-like” instructions
    - Contains commonly used mathematical operations
  - Compiler convert to assembly language

- **Interpreter**
  - Execute high-level language programs without compilation

1.8 History of Java

- **Java**
  - Originally developed for intelligent consumer-electronic devices by Sun Microsystems
  - Goals: small, fast, reliable, and portable programs
  - Then used for creating Web pages with dynamic content
  - Now also used for:
    - Developing large-scale enterprise applications
    - Enhancing WWW server functionality
    - Providing applications for consumer devices (cell phones, etc.)
    - Grid computing and Web services

1.9 Java Class Libraries

- **Classes**
  - Include methods that perform tasks
    - Return information after task completion
  - Used to build Java programs

- **Java contains class libraries**
  - Known as Java APIs (Application Programming Interfaces)
### 1.13 Basics of a Typical Java 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**
    - Verifier ensures bytecodes do not violate security requirements
  - **Execute**
    - Interpreter translates bytecodes into machine language

---

**The edit-compile-execute cycle**

- **Source file**
- **Class file**
- **Editor**
- **Compiler (javac)**
- **Virtual machine (java)**

---

**Summary**

- Computer: set of hardware resources to process programs
- Computer programs: set of instructions to conduct applications
- Hardware: physical devices in a computer
- Software: all sorts of programs
- Operating System: controls hardware devices
- Compiler: translates user readable programs into machine readable programs
- Interpreter: interprets programs without compilation
Part 3
Introduction to Java Applications

2.1 Introduction

• In this chapter
  – Introduce examples to illustrate features of Java
  – Two program styles - applications and applets
2.2 A First Program in Java: Printing a Line of Text

- Application
  - Program that executes using the `java` interpreter
- Sample program
  - Show program, then analyze each line

```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 class Welcome1

Welcome to Java Programming!
```
2.2  A First Program in Java: Printing a Line of Text

1  // Fig. 2.1: Welcome1.java

– Comments start with: //</br>
   • Comments ignored during program execution</br>
   • Document and describe code</br>
   • Provides code readability</br>
– Traditional comments: /* ... */</br>/* This is a traditional comment. It can be split over many lines */</br>

2  // Text-printing program.</br>– Another line of comments</br>– Note: line numbers not part of program, added for reference

2.2  A Simple Program: Printing a Line of Text

3  

– Blank line</br>  • Makes program more readable</br>  • Blank lines, spaces, and tabs are white-space characters</br>    – Ignored by compiler</br>

4  public class Welcome1 {

– Begins class declaration for class Welcome1</br>  • Every Java program has at least one user-defined class</br>  • Keyword: words reserved for use by Java</br>    – class keyword followed by class name</br>  • Naming classes: capitalize every word</br>    – SampleClassName
2.2 A Simple Program: Printing a Line of Text

- Name of class called identifier
  - Series of characters consisting of letters, digits, underscores (_), and dollar signs ($)
  - Does not begin with a digit, has no spaces
  - Examples: `Welcome1`, `$value`, `_value`, `button7`
  - `7button` is invalid
  - Java is case sensitive (capitalization matters)
    - `a1` and `A1` are different
- For chapters 2 to 7, use `public` keyword
  - Certain details not important now
  - Mimic certain features, discussions later

```java
public class Welcome1 {
    // Saving files
    // File name must be class name with .java extension
    // `Welcome1.java`
    // Left brace {
    // Begins body of every class
    // Right brace ends declarations (line 13)

    public static void main(String args[]) {
        // Part of every Java application
        // Applications begin executing at `main`
        // Parenthesis indicate `main` is a method (ch. 6)
        // Java applications contain one or more methods
    }
}
```
2.2 A Simple Program: Printing a Line of Text

- Exactly one method must be called \texttt{main}
- Methods can perform tasks and return information
  - \texttt{void} means \texttt{main} returns no information
  - For now, mimic \texttt{main}'s first line

```
7 public static void main( String args[] )

8 {

  - Left brace begins body of method declaration
  - Ended by right brace } (line 11)
```

- Instructs computer to perform an action
  - Prints string of characters
    - String - series characters inside double quotes
    - White-spaces in strings are not ignored by compiler
- \texttt{System.out}
  - Standard output object
  - Print to command window (i.e., MS-DOS prompt)
- Method \texttt{System.out.println}
  - Displays line of text
  - Argument inside parenthesis
- This line known as a statement
  - Statements must end with semicolon ;
2.2 A Simple Program: Printing a Line of Text

• Compiling a program
  – Open a command prompt window, go to directory where program is stored
  – Type javac Welcome1.java
  – If no errors, Welcome1.class created
    • Has bytecodes that represent application
    • Bytecodes passed to Java interpreter
2.2 A Simple Program: Printing a Line of Text

• Executing a program
  – Type `java Welcome1`
    • Interpreter loads `.class` file for class `Welcome1`
    • `.class` extension omitted from command
  – Interpreter calls method `main`

Fig. 2.2 Executing Welcome1 in a Microsoft Windows 2000 Command Prompt.

2.3 Modifying Our First Java Program

• Modify example in Fig. 2.1 to print same contents using different code
2.3 Modifying Our First Java Program

• Modifying programs
  – Welcome2.java (Fig. 2.3) produces same output as Welcome1.java (Fig. 2.1)
  – Using different code

```java
9 System.out.print( "Welcome to ");
10 System.out.println( "Java Programming!");
```

– Line 9 displays “Welcome to ” with cursor remaining on printed line
– Line 10 displays “Java Programming! ” on same line with cursor on next line
2.3 Modifying Our First Java Program

• Newline characters (\n)
  – Interpreted as “special characters” by methods
    System.out.println and System.out.println
  – Indicates cursor should be on next line
  – Welcome3.java (Fig. 2.4)

  ```java
  System.out.println("Welcome\nto\nJava\nProgramming!");
  ```

  – Line breaks at \n
• Usage
  – Can use in System.out.println or
    System.out.print to create new lines
    • System.out.println("Welcome\nto\nJava\nProgramming!");

// Fig. 2.4: Welcome3.java
// Printing multiple lines of text with a single statement.
public class Welcome3 {
  // main method begins execution of Java application
  public static void main( String args[] )
  {
    System.out.println("Welcome\nto\nJava\nProgramming!");
  }
  // end method main
  // end class Welcome3

Notice how a new line is output for each \n escape sequence.
2.3 Modifying Our First Java Program

Escape characters
- Backslash (\)
- Indicates special characters be output

<table>
<thead>
<tr>
<th>Escape sequence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n</td>
<td>Newline. Position the screen cursor at the beginning of the next line.</td>
</tr>
<tr>
<td>\t</td>
<td>Horizontal tab. Move the screen cursor to the next tab stop.</td>
</tr>
<tr>
<td>\r</td>
<td>Carriage return. Position the screen cursor at the beginning of the current line; do not advance to the next line. Any characters output after the carriage return overwrite the characters previously output on that line.</td>
</tr>
<tr>
<td>\</td>
<td>Backslash. Used to print a backslash character.</td>
</tr>
</tbody>
</table>
| "               | Double quote. Used to print a double-quote character. For example, 

```java
System.out.println("\"in quotes\"" );
```

displays "in quotes"

Fig. 2.5 Some common escape sequences.

2.4 Displaying Text in a Dialog Box

- Display
  - Most Java applications use windows or a dialog box
    - We have used command window
  - Class JOptionPane allows us to use dialog boxes

- Packages
  - Set of predefined classes for us to use
  - Groups of related classes called packages
    - Group of all packages known as Java class library or Java applications programming interface (Java API)
    - JOptionPane is in the javax.swing package
      - Package has classes for using Graphical User Interfaces (GUIs)
2.4 Displaying Text in a Dialog Box

- Web browsers display Web pages in their own windows
- E-mail programs enable a user to edit and read mails in windows
- Programs display important messages to the user in dialog boxes

2.4 Displaying Text in a Dialog Box

- Upcoming program
  - Application that uses dialog boxes
  - Explanation will come afterwards
  - Demonstrate another way to display output
  - Packages, methods and GUI
// Fig. 2.6: Welcome4.java
// Printing multiple lines in a dialog box.

// Java packages
import javax.swing.JOptionPane;  // program uses JOptionPane

public class Welcome4 {
    public static void main( String args[] )
    {
        JOptionPane.showMessageDialog(
            null, "Welcome
to
Java
Programming!" );
        System.exit( 0 );  // terminate application with window
    } // end method main
} // end class Welcome4

2.4 Displaying Text in a Dialog Box

– Lines 1-2: comments as before

// Java packages

– Two groups of packages in Java API
– Core packages
  • Begin with java
  • Included with Java 2 Software Development Kit
– Extension packages
  • Begin with javax
  • New Java packages

import javax.swing.JOptionPane;  // program uses JOptionPane

– import declarations
  • Used by compiler to identify and locate classes used in Java programs
  • Tells compiler to load class JOptionPane from javax.swing package
2.4 Displaying Text in a Dialog Box

– Lines 6-11: Blank line, begin class `Welcome` and `main`

```java
12 13 JOptionPane.showMessageDialog(
    null, "Welcome\n\nJava\nProgramming!" );
```

– Call method `showMessageDialog` of class `JOptionPane`

  • Requires two arguments
  • Multiple arguments separated by commas (,)
  • For now, first argument always `null`
  • Second argument is string to display

– `showMessageDialog` is a `static` method of class `JOptionPane`

  • `static` methods called using class name, dot (.) then method name

2.4 Displaying Text in a Dialog Box

– All statements end with ;

  • A single statement can span multiple lines
  • Cannot split statement in middle of identifier or string

– Executing lines 12 and 13 displays the dialog box

  • Automatically includes an **OK** button
    – Hides or dismisses dialog box
  • Title bar has string **Message**
2.4 Displaying Text in a Dialog Box

- Calls `static` method `exit` of class `System`
  - Terminates application
    - Use with any application displaying a GUI
  - Because method is `static`, needs class name and dot (.)
  - Identifiers starting with capital letters usually class names
- Argument of 0 means application ended successfully
  - Non-zero usually means an error occurred
- Class `System` part of package `java.lang`
  - No import declaration needed
  - `java.lang` automatically imported in every Java program
- Lines 17-19: Braces to end `welcome4` and `main`

2.5 Another Java Application: Adding Integers

- Upcoming program
  - Use input dialogs to input two values from user
  - Use message dialog to display sum of the two values
public class Addition {
    public static void main(String[] args) {
        String firstNumber; // first string entered by user
        String secondNumber; // second string entered by user
        int number1; // first number to add
        int number2; // second number to add
        int sum; // sum of number1 and number2

        firstNumber = JOptionPane.showInputDialog("Enter first integer");
        secondNumber = JOptionPane.showInputDialog("Enter second integer");

        number1 = Integer.parseInt(firstNumber);
        number2 = Integer.parseInt(secondNumber);

        sum = number1 + number2;

        JOptionPane.showMessageDialog(null, "The sum is " + sum, "Results", JOptionPane.PLAIN_MESSAGE);
        System.exit(0); // terminate application with window
    }
}

Add first integer as a String, assign to firstNumber.

Add, place result in sum.

Convert strings to integers.

Program output
2.5 Another Java Application: Adding Integers

5 import javax.swing.JOptionPane; // program uses JOptionPane

– Location of JOptionPane for use in the program

7 public class Addition {

– Begins public class Addition
  • Recall that file name must be Addition.java
  – Lines 10-11: main

12 String firstNumber;   // first string entered by user
13 String secondNumber;  // second string entered by user

– Declaration
  • firstNumber and secondNumber are variables

String firstNumber, secondNumber;

– Variables
  • Location in memory that stores a value
    – Declare with name and type before use
  • firstNumber and secondNumber are of type String
    (package java.lang)
    – Hold strings
  • Variable name: any valid identifier
  • Declarations end with semicolons ;

– Can declare multiple variables of the same type at a time
  – Use comma separated list
– Can add comments to describe purpose of variables
2.5 Another Java Application: Adding Integers

- Declares variables `number1`, `number2`, and `sum` of type `int`
  - `int` holds integer values (whole numbers): i.e., 0, –4, 97
  - Types `float` and `double` can hold decimal numbers
  - Type `char` can hold a single character: i.e., x, $, \n, 7
  - Primitive types - more in Chapter 4

```java
15 int number1;          // first number to add
16 int number2;          // second number to add
17 int sum;              // sum of number1 and number2
```

- Reads `String` from the user, representing the first number to be added
  - Method `JOptionPane.showInputDialog` displays the following:
    - Message called a prompt - directs user to perform an action
    - Argument appears as prompt text
    - If wrong type of data entered (non-integer) or click `Cancel`, error occurs

```java
20 firstNumber = JOptionPane.showInputDialog( "Enter first integer" );
```
2.5 Another Java Application: Adding Integers

- Result of call to `showInputDialog` given to `firstNumber` using assignment operator =
  - Assignment statement
  - = binary operator - takes two operands
    - Expression on right evaluated and assigned to variable on left
  - Read as: `firstNumber` gets value of `JOptionPane.showInputDialog( "Enter first integer" )`

```java
firstNumber = JOptionPane.showInputDialog( "Enter first integer" );
```

- Similar to previous statement
  - Assigns variable `secondNumber` to second integer input

```java
secondNumber = JOptionPane.showInputDialog( "Enter second integer" );
```

- Method `Integer.parseInt`
  - Converts String argument into an integer (type `int`)
    - Class `Integer` in `java.lang`
  - Integer returned by `Integer.parseInt` is assigned to variable `number1` (line 27)
    - Remember that `number1` was declared as type `int`
  - Line 28 similar

```java
number1 = Integer.parseInt( firstNumber );
number2 = Integer.parseInt( secondNumber );
```
2.5 Another Java Application: Adding Integers

Assignment statement
- Calculates sum of number1 and number2 (right hand side)
- Uses assignment operator = to assign result to variable sum
- Read as: sum gets the value of number1 + number2
- number1 and number2 are operands

```
    sum = number1 + number2;
```

- Use showMessageDialog to display results
- "The sum is " + sum
  - Uses the operator + to "add" the string literal "The sum is" and sum
  - Concatenation of a String and another type
  - Results in a new string
  - If sum contains 117, then "The sum is " + sum results in the new string "The sum is 117"
  - Note the space in "The sum is"
  - More on strings in Chapter 11
2.5 Another Java Application: Adding Integers

Different version of `showMessageDialog`
- Requires four arguments (instead of two as before)
- First argument: `null` for now
- Second: string to display
- Third: string in title bar
- Fourth: type of message dialog with icon
  - Line 35 no icon: `JOptionPane.PLAIN_MESSAGE`

```java
JOptionPane.showMessageDialog( null, "The sum is " + sum, "Results", JOptionPane.PLAIN_MESSAGE );
```

<table>
<thead>
<tr>
<th>Message dialog type</th>
<th>Icon</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>JOptionPane.ERROR_MESSAGE</code></td>
<td></td>
<td>Displays a dialog that indicates an error to the user.</td>
</tr>
<tr>
<td><code>JOptionPane.INFORMATION_MESSAGE</code></td>
<td></td>
<td>Displays a dialog with an informational message to the user. The user can simply dismiss the dialog.</td>
</tr>
<tr>
<td><code>JOptionPane.WARNING_MESSAGE</code></td>
<td></td>
<td>Displays a dialog that warns the user of a potential problem.</td>
</tr>
<tr>
<td><code>JOptionPane.QUESTION_MESSAGE</code></td>
<td></td>
<td>Displays a dialog that poses a question to the user. The dialog normally requires a response, such as clicking on a Yes or a No button.</td>
</tr>
<tr>
<td><code>JOptionPane.PLAIN_MESSAGE</code></td>
<td>no icon</td>
<td>Displays a dialog that simply contains a message, with no icon.</td>
</tr>
</tbody>
</table>

Fig. 2.12  `JOptionPane` constants for message dialogs.
2.6 Memory Concepts

• Variables
  – Every variable has a name, a type, a size and a value
    • Name corresponds to location in memory
  – When new value is placed into a variable, replaces (and destroys) previous value
  – Reading variables from memory does not change them

• Visual Representation
  – Sum = 0; number1 = 1; number2 = 2;
    
    ![Sum 0]

  – Sum = number1 + number2; after execution of statement
    
    ![Sum 3]
2.7 Arithmetic

• Arithmetic calculations used in most programs
  – Usage
    • * for multiplication
    • / for division
    • +, -
    • No operator for exponentiation (more in Chapter 5)
  – Integer division truncates remainder
    7 / 5 evaluates to 1
  – Remainder operator % returns the remainder
    7 % 5 evaluates to 2

2.7 Arithmetic

• Operator precedence
  – Some arithmetic operators act before others (i.e., multiplication before addition)
    • Use parenthesis when needed
  – Example: Find the average of three variables a, b and c
    • Do not use: a + b + c / 3
    • Use: (a + b + c) / 3
  – Follows PEMDAS
    • Parentheses, Exponents, Multiplication, Division, Addition, Subtraction
2.7 Arithmetic

<table>
<thead>
<tr>
<th>Operator(s)</th>
<th>Operation(s)</th>
<th>Order of evaluation (precedence)</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>Evaluated first. If there are several of this type of operator, they are evaluated from left to right.</td>
</tr>
<tr>
<td>/</td>
<td>Division, Remainder</td>
<td>Evaluated next. If there are several of this type of operator, they are evaluated from left to right.</td>
</tr>
<tr>
<td>+</td>
<td>Addition</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 2.17 Precedence of arithmetic operators.

2.8 Decision Making: Equality and Relational Operators

- `if` control statement
  - Simple version in this section, more detail later
  - If a condition is true, then the body of the `if` statement executed
    - 0 interpreted as false, non-zero is true
  - Control always resumes after the `if` structure
  - Conditions for `if` statements can be formed using equality or relational operators (next slide)

```plaintext
if ( condition )
    statement executed if condition true
```

- No semicolon needed after condition
  - Else conditional task not performed
2.8 Decision Making: Equality and Relational Operators

- Upcoming program uses if statements
  - Discussion afterwards

<table>
<thead>
<tr>
<th>Standard algebraic equality or relational operator</th>
<th>Java equality or relational operator</th>
<th>Example of Java condition</th>
<th>Meaning of Java condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equality operators</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>==</td>
<td>x == y</td>
<td>x is equal to y</td>
</tr>
<tr>
<td>!=</td>
<td>!=</td>
<td>x != y</td>
<td>x is not equal to y</td>
</tr>
<tr>
<td>Relational operators</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>&gt;</td>
<td>x &gt; y</td>
<td>x is greater than y</td>
</tr>
<tr>
<td>&lt;</td>
<td>&lt;</td>
<td>x &lt; y</td>
<td>x is less than y</td>
</tr>
<tr>
<td>≥</td>
<td>&gt;=</td>
<td>x &gt;= y</td>
<td>x is greater than or equal to y</td>
</tr>
<tr>
<td>≤</td>
<td>&lt;=</td>
<td>x &lt;= y</td>
<td>x is less than or equal to y</td>
</tr>
</tbody>
</table>

Fig. 2.19 Equality and relational operators.

```java
// Fig. 2.20: Comparison.java
// Compare integers using if statements, relational operators
// and equality operators.
// Java packages
import javax.swing.JOptionPane;

public class Comparison {
  public static void main( String args[] )
  {
    String firstNumber;   // first string entered by user
    String secondNumber;  // second string entered by user
    String result;        // a string containing the output

    int number1;          // first number to compare
    int number2;          // second number to compare

    // read first number from user as a string
    firstNumber = JOptionPane.showInputDialog( "Enter first integer:" );

    // read second number from user as a string
    secondNumber = JOptionPane.showInputDialog( "Enter second integer:" );

    // convert numbers from type String to type int
    number1 = Integer.parseInt( firstNumber );
    number2 = Integer.parseInt( secondNumber );

    // initialize result to empty String
    result = "";
```
Test for equality, create new string, assign to \texttt{result}.

\begin{verbatim}
if ( number1 == number2 )
    result = result + number1 + " == " + number2;

if ( number1 != number2 )
    result = result + number1 + " != " + number2;

if ( number1 < number2 )
    result = result + \texttt{\textbackslash n} + number1 + " < " + number2;

if ( number1 > number2 )
    result = result + \texttt{\textbackslash n} + number1 + " > " + number2;

if ( number1 <= number2 )
    result = result + \texttt{\textbackslash n} + number1 + " <= " + number2;

if ( number1 >= number2 )
    result = result + \texttt{\textbackslash n} + number1 + " >= " + number2;

// Display results
JOptionPane.showMessageDialog( null, result, \textquote{Comparison Results},
JOptionPane.INFORMATION_MESSAGE );

System.exit( 0 ); // terminate application
\end{verbatim}

Notice use of \texttt{JOptionPane.INFORMATION_MESSAGE}

Program Output

\begin{itemize}
\item \texttt{Input}
\item \texttt{Comparison Results}
\end{itemize}
2.8 Decision Making: Equality and Relational Operators

– Lines 1-12: Comments, import JOptionPane, begin class Comparison and main
– Lines 13-18: declare variables
  • Can use comma-separated lists instead:

```
13 String firstNumber,
14   secondNumber,
15   result;
```

– Lines 21-30: obtain user-input numbers and parses input string into integer variables

```java
result = "";
```

– Initialize result with empty string

```java
if ( number1 == number2 )
result = result + number1 + " == " + number2;
```

– if statement to test for equality using (==)
  • If variables equal (condition true)
    – result concatenated using + operator
    – result = result + other strings
    – Right side evaluated first, new string assigned to result
  • If variables not equal, statement skipped
2.8 Decision Making: Equality and Relational Operators

- Lines 37-50: other if statements testing for less than, more than, etc.
  - If `number1 = 123` and `number2 = 123`
    - Line 34 evaluates true (if `number1 == number2`)
    - Because `number1` equals `number2`
    - Line 40 evaluates false (if `number1 < number2`)
    - Because `number1` is not less than `number2`
    - Line 49 evaluates true (if `number1 >= number2`)
    - Because `number1` is greater than or equal to `number2`

- Lines 53-54: `result` displayed in a dialog box using `showMessageDialog`

2.8 Decision Making: Equality and Relational Operators

- Precedence of operators
  - All operators except for `=` (assignment) associates from left to right
  - For example: `x = y = z` is evaluated `x = (y = z)`

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>+   -   %</td>
<td>left to right</td>
<td>multiplicative</td>
</tr>
<tr>
<td>&lt;   &lt;= &gt;   &gt;=</td>
<td>left to right</td>
<td>relational</td>
</tr>
<tr>
<td>==  !=  ===</td>
<td>left to right</td>
<td>equality</td>
</tr>
<tr>
<td>=   !=</td>
<td>right to left</td>
<td>assignment</td>
</tr>
</tbody>
</table>

Fig. 2.21 Precedence and associativity of the operators discussed so far.
Summary

• Simple Java programs
• Input/output constructs
• Dialog boxes
• Memory concepts and variables
• Arithmetic, equality and relational operators
• Precedences and associativity
Part 4
Introduction to Java Applets

3.1 Introduction

• Applet
  – Program that runs in
    • appletviewer (test utility for applets)
    • Web browser (IE, Communicator)
  – Browser that executes applet is called applet container.
  – Executes when HTML (Hypertext Markup Language)
    document containing applet is opened and downloaded
  – Applications run in command windows

• Notes
  – Mimic several features of Chapter 2 to reinforce them
  – Focus on fundamental programming concepts first
    • Explanations will come later
3.2 Sample Applets from the Java 2 Software Development Kit

- **Sample Applets**
  - Provided in Java 2 Software Development Kit (J2SDK)
  - Source code included (.java files)
    - Study and mimic source code to learn new features
    - All programmers begin by mimicking existing programs
  - Located in `demo` directory of J2SDK install
  - Can download demos and J2SDK from [java.sun.com/j2se/1.4.1/](http://java.sun.com/j2se/1.4.1/)

3.2 Sample Applets from the Java 2 Software Development Kit

- **Running applets**
  - In command prompt, change to demo subdirectory of applet
    
    ```
    cd c:\j2sdk1.4.1\demo\applets
    cd appletDirectoryName
    ```
  - There will be an HTML file used to execute applet
  - Type `appletviewer example1.html`
    - `appletviewer` loads the html file specified as its command-line argument
    - From the HTML file, determines which applet to load (more section 3.3)
  - Applet will run, **Reload** and **Quit** commands under **Applet** menu
### 3.2 Sample Applets from the Java 2 Software Development Kit

- You start as player "X"

**Fig. 3.2** Sample execution of applet TicTacToe.

**Fig. 3.4** Sample execution of applet DrawTest.

- Drag the mouse pointer in the white area to draw.
- Select the drawing color by clicking the circle for the color you want. These GUI components are commonly known as radio buttons.
- Select the shape to draw by clicking the down arrow, then clicking Lines or Points. This GUI component is commonly known as a combo box, choice or drop-down list.
3.2 Sample Applets from the Java 2 Software Development Kit
• Demonstrates 2D drawing capabilities built into Java2

Click a tab to select a two-dimensional graphics demo.

Try changing the options to see their effect on the demonstration.

3.3 Simple Java Applet: Drawing a String

• Now, create applets of our own
  – Take a while before we can write applets like in the demos
  – Cover many of same techniques

• Upcoming program
  – Create an applet to display "Welcome to Java Programming!"
  – Show applet and HTML file, then discuss them line by line
import allows us to use predefined classes (allowing us to use applets and graphics, in this case).

extends allows us to inherit the capabilities of class JApplet.

Method paint is guaranteed to be called in all applets. Its first line must be defined as above.

3.3 Simple Java Applet: Drawing a String

– Comments
  • Name of source code and description of applet

import java.awt.Graphics;  // import class Graphics
import javax.swing.JApplet;  // import class JApplet

– Import predefined classes grouped into packages
  • import declarations tell compiler where to locate classes used
  • When you create applets, import the JApplet class (package javax.swing)
  • import the Graphics class (package java.awt) to draw graphics
    – Can draw lines, rectangles ovals, strings of characters
  • import specifies directory structure
3.3 Simple Java Applet: Drawing a String

- Applets have at least one class declaration (like applications)
  - Rarely create classes from scratch
    - Use pieces of existing classes
    - Inheritance - create new classes from old ones (ch. 9)

```
public class WelcomeApplet extends JApplet {

- Begins class declaration for class WelcomeApplet
  - Keyword class then class name
- extends followed by class name
  - Indicates class to extend (JApplet)
    - JApplet: superclass (base class)
    - WelcomeApplet: subclass (derived class)
  - WelcomeApplet now has methods and data of JApplet
```

3.3 Simple Java Applet: Drawing a String

```
public class WelcomeApplet extends JApplet {

- Class JApplet defined for us
  - Someone else defined "what it means to be an applet"
    - Applets require over 200 methods!
  - extends JApplet
    - Inherit methods, do not have to declare them all
  - Do not need to know every detail of class JApplet
```
3.3 Simple Java Applet: Drawing a String

---

8 public class WelcomeApplet extends JApplet {

- Class WelcomeApplet is a blueprint
  - appletviewer or browser creates an object of class WelcomeApplet
    - Keyword public required
    - File can only have one public class
    - public class name must be file name

---

11 public void paint( Graphics g )

- Our class inherits method paint from JApplet
  - By default, paint has empty body
  - Override (redefine) paint in our class

- Methods paint, init, and start
  - Guaranteed to be called automatically
  - Our applet gets "free" version of these by inheriting from JApplet
    - Free versions have empty body (do nothing)
    - Every applet does not need all three methods
      - Override the ones you need

- Applet container “draws itself” by calling method paint
3.3 Simple Java Applet: Drawing a String

11 \texttt{public void paint( Graphics g )}

- Method \texttt{paint}
  - Lines 11-19 are the declaration of \texttt{paint}
  - Draws graphics on screen
  - \texttt{void} indicates \texttt{paint} returns nothing when finishes task
  - Parenthesis define parameter list - where methods receive data to perform tasks
    - Normally, data passed by programmer, as in \texttt{JOptionPane.showMessageDialog}
  - \texttt{paint} gets parameters automatically
    - \texttt{Graphics} object used by \texttt{paint}
  - Mimic \texttt{paint}'s first line

14 \texttt{super.paint( g );}

- Calls version of method \texttt{paint} from superclass \texttt{JApplet}
- Should be first statement in every applet’s \texttt{paint} method

17 \texttt{g.drawString( "Welcome to Java Programming!", 25, 25 );}

- Body of \texttt{paint}
  - Method \texttt{drawString} (of class \texttt{Graphics})
  - Called using \texttt{Graphics} object \texttt{g} and dot (.)
  - Method name, then parenthesis with arguments
    - First argument: \texttt{String} to draw
    - Second: x coordinate (in pixels) location
    - Third: y coordinate (in pixels) location
- Java coordinate system
  - Measured in pixels (picture elements)
  - Upper left is (0,0)
3.3 Simple Java Applet: Drawing a String

- Running the applet
  - Compile
    - `javac WelcomeApplet.java`
    - If no errors, bytecodes stored in `WelcomeApplet.class`
  - Create an HTML file
    - Loads the applet into `appletviewer` or a browser
    - Ends in `.htm` or `.html`
  - To execute an applet
    - Create an HTML file indicating which applet the browser (or `appletviewer`) should load and execute

```html
<html>
  <applet code = "WelcomeApplet.class" width = "300" height = "45">
  </applet>
</html>
```

- Simple HTML file (`WelcomeApplet.html`)
  - Usually in same directory as `.class` file
  - Remember, `.class` file created after compilation
- HTML codes (tags)
  - Usually come in pairs
  - Begin with `<` and end with `>`
- Lines 1 and 4 - begin and end the HTML tags
- Line 2 - begins `<applet>` tag
  - Specifies code to use for applet
  - Specifies `width` and `height` of display area in pixels
- Line 3 - ends `<applet>` tag
3.3 Simple Java Applet: Drawing a String

- **appletviewer** only understands `<applet>` tags
  - Ignores everything else
  - Minimal browser
- Executing the applet
  - `appletviewer WelcomeApplet.html`
  - Perform in directory containing `.class` file

```html
<html>
  <applet code = "WelcomeApplet.class" width = "300" height = "45">
  </applet>
</html>
```

3.3 Simple Java Applet: Drawing a String

- Running the applet in a Web browser
3.4 Drawing Strings and Lines

- More applets
  - First example
    - Display two lines of text
    - Use `drawString` to simulate a new line with two `drawString` statements
  - Second example
    - Method `g.drawLine(x1, y1, x2, y2)`
      - Draws a line from (x1, y1) to (x2, y2)
      - Remember that (0, 0) is upper left
    - Use `drawLine` to draw a line beneath and above a string

```java
// Fig. 3.9: WelcomeApplet2.java
// Displaying multiple strings in an applet.
// Java packages
import java.awt.Graphics; // import class Graphics
import javax.swing.JApplet; // import class JApplet

public class WelcomeApplet2 extends JApplet {

    public void paint(Graphics g) {
        // call superclass version of method paint
        super.paint(g);

        // draw two Strings at different locations
        g.drawString("Welcome to", 25, 25);
        g.drawString("Java Programming!", 25, 40);
    }
    } // end method paint

} // end class WelcomeApplet2
```

The two `drawString` statements simulate a newline. In fact, the concept of lines of text does not exist when drawing strings.
```java
// Fig. 3.11: WelcomeLines.java
// Displaying text and lines

// Java packages
import java.awt.Graphics;    // import class Graphics
import javax.swing.JApplet;  // import class JApplet

public class WelcomeLines extends JApplet {

    // draw lines and a string on applet's background
    public void paint( Graphics g ) {
        super.paint( g );

        // draw horizontal line from (15, 10) to (210, 10)
        g.drawLine( 15, 10, 210, 10 );

        // draw horizontal line from (15, 30) to (210, 30)
        g.drawLine( 15, 30, 210, 30 );

        // draw String between lines at location (25, 25)
        g.drawString( "Welcome to Java Programming!", 25, 25 );

    } // end method paint

} // end class WelcomeLines
```

Draw horizontal lines with `drawLine` (endpoints have same y coordinate).
3.4 Drawing Strings and Lines

- Method `drawLine` of class `Graphics`
  - Takes as arguments `Graphics` object and line’s end points
  - x and y coordinate of first endpoint
  - x and y coordinate of second endpoint
3.5 Adding Floating-Point Numbers

- Next applet
  - Mimics application for adding two integers (Fig 2.9)
    - This time, use floating point numbers (numbers with a decimal point)
      - Using primitive types
        - `double` – double precision floating-point numbers
        - `float` – single precision floating-point numbers
  - Show program, then discuss

```java
// Fig. 3.13: AdditionApplet.java
// Adding two floating-point numbers.

// Java packages
import java.awt.Graphics;   // import class Graphics
import javax.swing.*;       // import package javax.swing

public class AdditionApplet extends JApplet {
    double sum;  // sum of values entered by user

    // initialize applet by obtaining values from user
    public void init()
    {
        String firstNumber;  // first string entered by user
        String secondNumber; // second string entered by user

        double number1;      // first number to add
        double number2;       // second number to add

        // obtain first number from user
        firstNumber = JOptionPane.showInputDialog("Enter first floating-point value");

        // obtain second number from user
        secondNumber = JOptionPane.showInputDialog("Enter second floating-point value");

        // convert numbers from type String to type double
        number1 = Double.parseDouble( firstNumber );
        number2 = Double.parseDouble( secondNumber );

        // calculate sum of numbers entered by user
        sum = number1 + number2;

        // display result
        System.out.println("Sum = "+sum);
    }

    public void paint(Graphics g)
    {
        g.drawString("Sum = "+sum, 320, 150);
    }
}
```
```java
// add numbers
sum = number1 + number2;

} // end method init

// draw results in a rectangle on applet's background
public void paint( Graphics g )
{
    // call superclass version of method paint
    super.paint( g );

    // draw rectangle starting from (15, 10) that is 270
    // pixels wide and 20 pixels tall
    g.drawRect( 15, 10, 270, 20);

    // draw results as a String at (25, 25)
    g.drawString( "The sum is " + sum, 25, 25 );

} // end method paint

} // end class AdditionApplet
```

`drawRect` takes the upper left coordinate, width, and height of the rectangle to draw.

---

**HTML file**

```html
<html>
<applet code = "AdditionApplet.class" width = "300" height = "65"
</applet>
</html>
```

---

**Program Output**

Applet loaded.

Enter first floating point value
45.5

OK Cancel

Enter second floating point value
72.37

OK Cancel

The sum is 117.87

Applet started.
3.5 Adding Floating-Point Numbers

– Lines 1-2: Comments

5 \texttt{import java.awt.Graphics; // import class Graphics}

– Line 5: imports class Graphics
  • import not needed if use full package and class name

public void paint ( java.awt.Graphics g )

6 \texttt{import javax.swing.*; // import package javax.swing}

– Line 8: specify entire javax.swing package
  • * indicates all classes in javax.swing are available
    – Includes JApplet and JOptionPane
    – Use JOptionPane instead of javax.swing.JOptionPane
  • * does not not load all classes
    – Compiler only loads classes it uses

8 public class AdditionApplet extends JApplet {

– Begin class declaration
  • Extend JApplet, imported from package javax.swing

9 double sum; // sum of values entered by user

– Field declaration
  • Each object of class gets own copy of the field
  • Declared in body of class, but not inside methods
    – Variables declared in methods are local variables
    – Can only be used in body of method
  • Fields can be used anywhere in class
  • Have default value (0.0 in this case)
  • Fields are also known as instance variables.

Explicitly initializing fields rather than relying on automatic initialization improves program readability.
3.5 Adding Floating-Point Numbers

- Primitive type `double`
  - Used to store floating point (decimal) numbers

```java
double sum; // sum of values entered by user
```

- Method `init`
  - Normally initializes fields and applet class
  - Guaranteed to be first method called in applet
  - First line must always appear as above
    - Returns nothing (`void`), takes no arguments

```java
public void init()
{

```
- Begins body of method `init`

```java
String firstNumber; // first string entered by user
String secondNumber; // second string entered by user
double number1;      // first number to add
double number2;       // second number to add
```

- Declare variables
- Two types of variables
  - Reference variables (called `references`)
    - Refer to objects (contain location in memory)
      - Objects defined in a class definition
      - Can contain multiple data and methods
    - `paint` receives a reference called `g` to a `Graphics` object
      - Reference used to call methods on the `Graphics` object
  - Primitive types (called `variables`)
    - Contain one piece of data
    - Boolean, char, byte, short, int, long, float, and double
3.5 Adding Floating-Point Numbers

- Distinguishing references and variables
  - If type is a class name, then reference
    - String is a class
    - firstNumber, secondNumber
  - If type a primitive type, then variable
    - double is a primitive type
    - number1, number2

```java
String firstNumber;    // first string entered by user
String secondNumber;  // second string entered by user
double number1;       // first number to add
double number2;       // second number to add
```

- Method `JOptionPane.showInputDialog`
  - Prompts user for input with string
  - Enter value in text field, click OK
    - If not of correct type, error occurs
    - In Chapter 15 learn how to deal with this
  - Returns string user inputs
  - Assignment statement to string
    - Lines 25-26: As above, assigns input to `secondNumber`

```java
firstNumber = JOptionPane.showInputDialog(
    "Enter first floating-point value");
```
3.5 Adding Floating-Point Numbers

- **static** method `Double.parseDouble`
  - Converts `String` argument to a `double`
  - Returns the `double` value
  - Remember static method syntax
    - `ClassName.methodName( arguments )`

- Assignment statement
  - `sum` a field, can use anywhere in class
    - Not defined in `init` but still used

```
number1 = Double.parseDouble( firstNumber );
number2 = Double.parseDouble( secondNumber );

sum = number1 + number2;
```
3.5 Adding Floating-Point Numbers

- Sends `drawString` message (calls method) to `Graphics` object using reference `g`
  - "The sum is" + `sum` - string concatenation
    - `sum` converted to a string
  - `sum` can be used, even though not defined in `paint`
    - `field`, can be used anywhere in class
    - Non-local variable in method `drawString`

```java
48 g.drawString("The sum is " + sum, 25, 25);
```

3.6 Java Applet Internet and World Wide Web Resources

- Many Java applet resources available
  - `java.sun.com/applets/`
  - Many resources and free applets
    - Has demo applets from J2SDK
  - Sun site developer.java.sun.com/developer
    - Tech support, discussion forums, training, articles, links, etc.
    - Registration required
  - `www.jars.com`
    - Rates applets, top 1, 5 and 25 percent
    - View best applets on web
Summary

Applets

• Program that runs in
  – appletviewer (test utility for applets)
  – Web browser (IE, Communicator)

• Browser that executes applet is called applet container.

• Executes when HTML (Hypertext Markup Language) document containing applet is opened and downloaded

• Applications run in command windows
Part 5
Program Analysis
Principles and Techniques

Front end

- Responsibilities:
  - Recognize legal programs
  - Report errors
  - Produce $il$
  - Preliminary storage map
  - Shape the code for the back end
  - Much of front end construction can be automated
Scanner

- Scanner
  - Maps characters into tokens – the basic unit of syntax
    
    \[ x = x + y; \]
    
    Becomes
    
    \[ <id,x> = <id,x> + <id,y> ; \]
  - character string for token is a lexeme
  - Typical tokens: number, id, +, -, *, /, do, end
  - Eliminates white space (tables, blanks, comments)
  - A key issue is speed
    
    use specialized recognizer (lex)

Specifying patterns

- A scanner must recognize various parts of the language’s syntax.

- Some parts are easy:
  - White space
    some combination of \(<\hspace{3pt}\) and tab

  - Keywords and operators
    specified as literal patterns – do, end

  - Comments
    opening and closing delimiters - /* … */
Specifying patterns

• Other parts are much harder:
  – *Identifiers*
    alphabetic followed by $k$ alphanumerics
  – *Special symbols*
    (, $, &, …)
  – *Numbers*
    integer – 0 or digit from 1-9 followed by digitals from 0-9
decimals – integer “.” digits from 0-9\nreals – (integer or decimal) “E” (+ or -) digits from 0-9\ncomplex – “(” real “,” real “)"

  – *We need a powerful notation to specify these patterns.*

---

**Definitions**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Union of $L$ and $M$</td>
<td>$L \cup M = {s \mid s \in L \text{ or } s \in M}$</td>
</tr>
<tr>
<td>written $L \cup M$</td>
<td></td>
</tr>
<tr>
<td>Concatenation of $L$ and $M$</td>
<td>$LM = {st \mid s \in L \text{ and } t \in M}$</td>
</tr>
<tr>
<td>written $LM$</td>
<td></td>
</tr>
<tr>
<td>Kleene closure of $L$</td>
<td>$L^* = \bigcup_{i=0}^{\infty} L^i$</td>
</tr>
<tr>
<td>written $L^*$</td>
<td></td>
</tr>
<tr>
<td>Positive closure of $L$</td>
<td>$L^+ = \bigcup_{i=1}^{\infty} L^i$</td>
</tr>
<tr>
<td>written $L^+$</td>
<td></td>
</tr>
</tbody>
</table>
Regular expressions

• Patterns are often specified as regular languages. Notations used to describe a regular language (or a regular set) include both regular expressions and regular grammars.

• **Regular expressions** (over an alphabet \( \Sigma \)):

1. \( \varepsilon \) is a RE denoting the set \( \{ \varepsilon \} \)

2. If \( a \in \Sigma \), then \( a \) is RE denoting \( \{a\} \)

• **Regular expressions** (over an alphabet \( \Sigma \)):

3. If \( r \) and \( s \) are REs, denoting \( L(r) \) and \( L(s) \), then:

   - \((r)\) is a RE denoting \( L(r) \)
   - \((r) \mid (s)\) is a RE denoting \( L(r) \cup L(s) \)
   - \((r) \cdot (s)\) is a RE denoting \( L(r) \cdot L(s) \)
   - \((r)^*\) is a RE denoting \( L(r)^* \)
   - \((r)^+\) is a RE denoting \( L(r)^+ \)

• If we adopt a precedence for operators, the extra parentheses can go away. We assume closure, then concatenation, the alternation as the order of precedence.
RE examples

• identifier

  - letter $\rightarrow$ (a | b | c | … | z | A | B | C | … | Z)
  - digit $\rightarrow$ (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)
  - id $\rightarrow$ letter (letter | digit)*

• numbers

  - integer $\rightarrow$ (+ | - | $\varepsilon$) (0 | 1 | 2 | 3 | … | 9) (digit)*
  - decimal $\rightarrow$ integer “.” (digit)*
  - real $\rightarrow$ (integer | decimal) $E$ (+ | -) (digit)*
  - Complex $\rightarrow$ “(”real “.” real “)”

RE examples

• Most programming language tokens can be described with regular expressions.

• We can use regular expressions to automatically build scanners.
Regular expressions

- Regular expressions represent languages
- Languages are sets of strings.
- Operations include *Kleene closure*, *concatenation*, and *union*.

<table>
<thead>
<tr>
<th>Regular expression</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>{“a“}</td>
</tr>
<tr>
<td>(a)</td>
<td>(b)</td>
</tr>
<tr>
<td>(a)(b)</td>
<td>{“ab“}</td>
</tr>
<tr>
<td>(a)*</td>
<td>{“““ , “a“ , “aa“, …}</td>
</tr>
<tr>
<td>(a)+</td>
<td>{“““ , “aa“, …}</td>
</tr>
</tbody>
</table>

Regular expressions

- We assume that *closure*, *concatenation*, *union* as the order of precedence.

\[
ab | cd* = (ab) | (c(d*))
\]

\[
a (bc)^* = \{“a“ , “abc“ , “abcbc“ , … \}
\]
Recognizers

- From a regular expression, we can construct a deterministic finite automan (dfa).

- Recognizer for the identifier:

```plaintext
Recognizers

• Identifier:
  - letter → (a | b | c | … | z | A | B | C | … | Z)
  - digit → (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)
  - id → letter (letter | digit)*
```

![Diagram of finite automaton for identifier recognition]
So what is hard?

• Language features that can cause problems:
  
  – *Reserved words*
    
    PL/I had no reserved words
    
    if then then then = else;
    
    else else = then;

  – *Significant blanks*
    
    FORTRAN and Algol68 ignore blanks
    
    do 10 I = 1,25
    
    do 10 I = 1.25

  – *String constants*
    
    special characters in strings
    
    newline, tab, quote, comment,

  – *Finite closures*
    
    some languages limit identifier lengths
    
    adds states to count length
    
    FORTRAN 66 \(\rightarrow\) 6 characters

• These problems can be swept under the rug (avoided) by intelligent language design.
Methods and parameters

- What is a lexical error?
  - 1234G6
  - Illegal character

- What should the scanner do?
  - Report the error
  - Try to connect it?

- Error connection techniques
  - Minimum distance corrections
  - Hard token recovery
  - Skip until match

Scanners

- A scanner separates input into tokens based on lexical analysis.

- Legal tokens are usually specified by regular expressions (REs).

- Regular expressions specify regular languages.
Limits of regular languages

• Not all languages are regular.

• You cannot construct dfa’s to recognize these languages:
  
  – \( L = \{ p^kq^k \} \)
  
  – \( L = \{ wcw^r \mid w \in \Sigma^* \} \)

• Note: neither of these is a regular expression!  
  (df a’s cannot count!)

Limits of regular languages

• But, this is a little subtle. You can construct dfa’s for:

  – Alternating 0’s and 1’s

    \( (\varepsilon \mid 1)(01)^* (\varepsilon \mid 0) \)

  – Sets of pairs of 0’s and 1’s

    \( (01 \mid 10)^+ \)
More regular languages

• Let’s look at another regular language – the set of strings containing an even number of zeros and an even number of ones.

• The regular expression is:

\[(00 \mid 11)^* ((01 \mid 10)(00 \mid 11)^*(01 \mid 10)(00 \mid 11)^*)^*\]

Summary

• Scanners
  – Break up input into tokens
  – Catch lexical errors
  – Difficulty affected by language design

• Scanner generators
  – Tokens specified by regular expressions
  – Construct \(DFA\) to recognize language
  – Highly efficient in practise
The role of the parser

- Parser:
  - Perform context-free syntax analysis
  - Guide the context-sensitive analysis
  - Construct an intermediate representation
  - Produce meaningful error messages
  - Attempt error correction

Syntax analysis

- *Context-free syntax is specified with a grammar.*

- Formally, a context-free grammar $G$ is a four-tuple $(T, NT, S, P)$
  
  - $T$ is the set of terminal symbols in the grammar. For our purposes, the set of terminals is equivalent to the set of tokens returned by the lexical analyzer.
Syntax analysis

— \( NT \) is a set of syntactic variables that denote sets of (sub)strings occurring in the language. These are used to impose a structure on the grammar.

— \( S \) is a distinguished nonterminal \((S \in NT)\) that denotes the entire set of strings in \( L(G) \). This is sometimes called a goal symbol. \( S \) cannot appear on the right hand side of some production.

— \( P \) is a set of productions that specify the way that terminals and non-terminals can be combined to form strings in the language. Each production must have a single non-terminal on its left hand side.

Syntax analysis

• Grammars are often written in BNF, or Backus-Naur Form

1. \(<\text{goal}> ::= <\text{expr}>\)
2. \(<\text{expr}> ::= <\text{expr}> <\text{op}> <\text{expr}>\)
3. \(\mid \text{number}\)
4. \(\mid \text{id}\)
5. \(<\text{op}> ::= +\)
6. \(\mid -\)
7. \(\mid *\)
8. \(\mid /\)
Syntax analysis

• This grammar gives simple expressions over numbers and identifiers.

• In a BNF for a grammar, we represent
  – Non-terminal with brackets or capital letters,
  – Terminals with typewriter font or underline,
  – Productions as in example

Why use context-free grammars?

• Many advantages:
  – Precise syntactic specification of a programming language
  – Easy to understand, avoids ad hoc definition
  – Easier to maintain, add new language features
  – Can automatically construct efficient parser
  – Parser construction reveals ambiguity, other difficulties
  – Imparts structure to language
  – Supports syntax-directed translation
Grammars for regular languages

• Can we place a restriction on the form of grammar to ensure that it describes a regular language?

• **Provable fact:**
  
  – For any RE \( r \), there is a grammar \( g \) such, that \( L(r) = L(g) \)

  – The grammar that generate regular sets are called *regular grammar*

---

Grammars for regular languages

• **Definition:**

  – In a regular grammar, all productions have one of two forms:

    1. \( A \to aA \)
    2. \( A \to a \)

  – Where \( A \) is a non-terminal and \( a \) is a terminal symbol.

  – These are also called *type 3 grammars* (Chomsky).
Scanning vs. Parsing

• *Where do we draw the line?*

  – `term \rightarrow [a-zA-z] ( [ a-zA-z ] | [ 0-9 ] )*`
  – `op \rightarrow + | - | * | /`
  – `expr \rightarrow (term op)* term`

• Regular expressions are used to classify.

  – identifiers, numbers, keywords

Scanning vs. Parsing

• Context-free grammars are used to count.

  – Brackets `\rightarrow ( )`, begin – end, if – then – else,

  – imposing structure - expressions

• Grammar for cc has 188 productions.
Derivations

• We can view the productions of a cfg as rewriting rules.

• Using a example:
  – \(<\text{goal}>\) \(\rightarrow\) \(<\text{expr}>\)
    \(\rightarrow\) \(<\text{expr}> <\text{op}> <\text{expr}>\)
    \(\rightarrow\) \(<\text{expr}> <\text{op}> <\text{expr}> <\text{op}> <\text{expr}>\)
    \(\rightarrow\) \(<\text{id,x}> <\text{op}> <\text{expr}> <\text{op}> <\text{expr}>\)
    \(\rightarrow\) \(<\text{id,x}> + <\text{expr}> <\text{op}> <\text{expr}>\)
    \(\rightarrow\) \(<\text{id,x}> + <\text{num,2}> <\text{op}> <\text{expr}>\)
    \(\rightarrow\) \(<\text{id,x}> + <\text{num,2}> * <\text{expr}>\)
    \(\rightarrow\) \(<\text{id,x}> + <\text{num,2}> * <\text{id,y}>\)

• We have derived the sentence \(x + 2 \times y\)

• We denote this \(<\text{goal}>\) \(\rightarrow\) \(\text{id} + \text{num} \times \text{id}\).

• Such a sequence of rewrites is a \textit{derivation} or a parse.

• The process of discovering a derivation is called \textit{parsing}.
Derivations

• At each step, we chose a non-terminal to replace.
• This choice can lead to different derivations.

• Two are of particular interest

  – Leftmost derivation
    the leftmost non-terminal is replaced at each step

  – Rightmost derivation
    the rightmost non-terminal is replaced at each step.

• The example was a leftmost derivation.

Rightmost Derivation

• For the string $x + 2 + y$:

  $<\text{goal}> \rightarrow <\text{expr}>
  \rightarrow <\text{expr}> <\text{op}> <\text{expr}>
  \rightarrow <\text{expr}> <\text{op}> <\text{id},y>
  \rightarrow <\text{expr}> <\text{op}> <\text{id},y>
  \rightarrow <\text{expr}> <\text{op}> <\text{expr}> * <\text{id},y>
  \rightarrow <\text{expr}> <\text{op}> <\text{num},2> * <\text{id},y>
  \rightarrow <\text{expr}> <\text{op}> <\text{num},2> * <\text{id},y>
  \rightarrow <\text{term}> + <\text{num},2> * <\text{id},y>
  \rightarrow <\text{id},x> + <\text{num},2> * <\text{id},y>

• Again, $<\text{goal}> \rightarrow \ldots \rightarrow \text{id} + \text{num} * \text{id}$. 
Let's look at the parse tree:

Treewalk evaluation would give the "wrong" answer:

\[(x + 2) \times y\] instead of \[x + (2 \times y)\]

These two derivations point out a problem with the grammar.

The grammar has no notion of precedence, or implied order of evaluation.
Precedence

• To add precedence takes additional machinery

1. \(<\text{goal}\> ::= \,<\text{expr}\>\)
2. \(<\text{expr}\> ::= \,<\text{expr}\> + \,<\text{term}\>\)
3. \(|<\text{expr}\> - <\text{term}\>|
4. \(|<\text{term}\>|
5. \,<\text{term}\> ::= \,<\text{term}\> * \,<\text{factor}\>\)
6. \(|<\text{term}\> / <\text{factor}\>|
7. \(|<\text{factor}\>|
8. \,<\text{factor}\> ::= \text{number}\)
9. \(|\text{id}\>|

Precedence

• This grammar enforces a precedence on the derivation

– terms must be derived from expressions

– forces the "correct" tree
Precedence

• Now, for the string $x + 2 * y$:

  \[
  \begin{array}{c}
  \text{<goal> } \Rightarrow \text{ <expr>} \\
  \Rightarrow \text{ <expr> + <term>} \\
  \Rightarrow \text{ <expr> + <term> * <factor>} \\
  \Rightarrow \text{ <expr> + <term> * <id,y>} \\
  \Rightarrow \text{ <expr> + <factor> * <id,y>} \\
  \Rightarrow \text{ <expr> + <num,2> * <id,y>} \\
  \Rightarrow \text{ <term> + <num,2> * <id,y>} \\
  \Rightarrow \text{ <factor> + <num,2> * <id,y>} \\
  \Rightarrow \text{ <id,x> + <num,2> * <id,y>} \\
  \end{array}
  \]

• Again, $\text{<goal> } \Rightarrow \ldots \Rightarrow \text{id + num * id}$, but this time, we build the desired tree.

Precedence

• \textit{This time}, we get the desired parse tree.

• \textit{Treewalk evaluation computes} $x + (2 * y)$.
Ambiguity

• If a grammar has multiple leftmost derivations for a single (terminal) word of the language, the grammar is *ambiguous*.

• Similarly, a grammar with multiple rightmost derivations for a single (terminal) word of the language is *ambiguous*.

• **Example**

  \[
  \langle \text{stmt} \rangle := \text{if \langle \text{expr} \rangle then \langle \text{stmt} \rangle} \\
  \quad \text{if \langle \text{expr} \rangle then \langle \text{stmt} \rangle else \langle \text{stmt} \rangle} \\
  \quad \text{other \ \text{stmts}}
  \]

Ambiguity

• Consider deriving the sentential form:

  – if \( E_1 \) then if \( E_2 \) then \( S_1 \) else \( S_2 \)

• It has two derivations.

• This ambiguity is purely grammatical.
• It is a *context-free* ambiguity.
Ambiguity

- We may be able to eliminate ambiguities by rearranging the grammar.

\[
\text{<stmt>} ::= \text{<ms>}
| \text{<us>}
\]

\[
\text{<ms>} ::= \text{if <expr> then <ms> else <ms>}
| \text{other stmts}
\]

\[
\text{<us>} ::= \text{if <expr> then <stmt>}
| \text{if <expr> then <ms> else <us>}
\]

Ambiguity

- This grammar generates the same language as the ambiguous grammar, but applies the common sense rule

- *match each else with the closest unmatched then*

- This is pretty clearly the language designer's intent.
Ambiguity

• Ambiguity generally refers to a confusion in the context-free specification.

• Context-sensitive confusions can arise from overloading.
  \[ a = f(17) \]

• In many Algol-like languages, \( f \) could be either a function or a subscripted variable.

Ambiguity

• Disambiguating this Statement requires context.
  – need values of declarations
  – not context free
  – really an issue of type

• Rather than complicate parsing, we will handle this separately.
Summary: Parser

- Parser performs context-free syntax analysis
- Recognizes structure over set of tokens
- Deals with ambiguity and precedences
4.2 Algorithms

- Algorithm
  - Series of actions in specific order
    - The actions executed
    - The order in which actions execute

- Program control
  - Specifying the order in which actions execute
    - Control structures help specify this order
4.3 Pseudocode

- Pseudocode
  - Informal language for developing algorithms
  - Not executed on computers
  - Helps developers “think out” algorithms

4.4 Control Structures

- Sequential execution
  - Program statements execute one after the other

- Transfer of control
  - Three control statements can specify order of statements
    - Sequence structure
    - Selection structure
    - Repetition structure

- Activity diagram
  - Models the workflow
    - Action-state symbols
    - Transition arrows
Fig 4.1 Sequence structure activity diagram.

<table>
<thead>
<tr>
<th>Java Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract assert boolean break byte</td>
</tr>
<tr>
<td>case catch char class continue</td>
</tr>
<tr>
<td>default do double else extends</td>
</tr>
<tr>
<td>final finally float for if</td>
</tr>
<tr>
<td>implements import instanceof int interface</td>
</tr>
<tr>
<td>long native new package private</td>
</tr>
<tr>
<td>protected public return short static</td>
</tr>
<tr>
<td>strictfp super switch synchronized this</td>
</tr>
<tr>
<td>throw throws transient try void</td>
</tr>
<tr>
<td>volatile while</td>
</tr>
</tbody>
</table>

*Keywords that are reserved, but not currently used*

*Fig. 4.2 Java keywords.*
4.4 Control Structures

- Java has a sequence structure “built-in”
- Java provides three selection structures
  - if
  - If…else
  - switch
- Java provides three repetition structures
  - while
  - do…while
  - do
- Each of these words is a Java keyword

4.5 if Single-Selection Statement

- Single-entry/single-exit control structure
- Perform action only when condition is true
- Action/decision programming model
4.6 if...else Selection Statement

- Perform action only when condition is true
- Perform different specified action when condition is false
- Conditional operator (?:)
- Nested if...else selection structures
4.7 while Repetition Statement

- Repeat action while condition remains true
4.8 Formulating Algorithms: Case Study 1 (Counter-Controlled Repetition)

- **Counter**
  - Variable that controls number of times set of statements executes

- **Average1.java** calculates grade averages
  - uses counters to control repetition
Set total to zero
Set grade counter to one

While grade counter is less than or equal to ten
   Input the next grade
   Add the grade into the total
   Add one to the grade counter

Set the class average to the total divided by ten
Print the class average

Fig. 4.6 Pseudocode algorithm that uses counter-controlled repetition to solve the class-average problem.

---

```java
// Fig. 4.7: Average1.java
// Class-average program with counter-controlled repetition.
import javax.swing.JOptionPane;

public class Average1 {

    public static void main( String args[] )
    {
        int total;          // sum of grades input by user
        int gradeCounter;   // number of grade to be entered next
        int grade;          // grade value
        int average;        // average of grades

        String gradeString; // grade typed by user

        // initialization phase
        total = 0;          // initialize total
        gradeCounter = 1;   // initialize loop counter

        // processing phase
        while ( gradeCounter <= 10 ) {  // loop 10 times
            gradeString = JOptionPane.showInputDialog(  
                "Enter integer grade: ");  
            grade = Integer.parseInt( gradeString );
        }
    }
}
```

Outline

Average1.java

gradeCounter is the counter
Continue looping as long as gradeCounter is less than or equal to 10
```java
    total = total + grade;            // add grade to total
    gradeCounter = gradeCounter + 1; // increment counter
}
} // end while

    // termination phase
    average = total / 10;  // integer division

    // display average of exam grades
    JOptionPane.showMessageDialog( null, "Class average is " + average,
        "Class Average", JOptionPane.INFORMATION_MESSAGE );
    System.exit( 0 );  // terminate the program
}
} // end main

} // end class Average1
```
4.9 Formulating Algorithms with Top-Down, Stepwise Refinement: Case Study 2 (Sentinel-Controlled Repetition)

- Sentinel value
  - Used to indicate the end of data entry
- Average2.java has indefinite repetition
  - User enters sentinel value (-1) to end repetition

```
Initialize total to zero
Initialize counter to zero

Input the first grade (possibly the sentinel)

While the user has not as yet entered the sentinel
  Add this grade into the running total
  Add one to the grade counter
  Input the next grade (possibly the sentinel)

If the counter is not equal to zero
  Set the average to the total divided by the counter
  Print the average
else
  Print “No grades were entered”
```

Fig. 4.8 Class-average problem pseudocode algorithm with sentinel-controlled repetition.
public class Average2 {

    public static void main( String args[] )
    {
        int total;           // sum of grades
        int gradeCounter;    // number of grades entered
        int grade;           // grade value

        double average;  // number with decimal point for average

        String gradeString;  // grade typed by user

        // initialization phase
        total = 0;         // initialize total
        gradeCounter = 0;  // initialize loop counter

        // processing phase
        // get first grade from user
        gradeString = JOptionPane.showInputDialog( "Enter Integer Grade or -1 to Quit:"

        // loop until sentinel value read from user
        while ( grade != -1 ) {
            total = total + grade;
            gradeCounter = gradeCounter + 1;

            // get next grade from user
            gradeString = JOptionPane.showInputDialog( "Enter Integer Grade or -1 to Quit:"

            // convert gradeString to int
            grade = Integer.parseInt( gradeString );

            // format numbers to nearest hundredth
        }

        // termination phase
        DecimalFormat twoDigits = new DecimalFormat( "0.00" );

        // if user entered at least one grade...
        if ( gradeCounter != 0 ) {
            // calculate average of all grades entered
            average = (double) total / gradeCounter;

            // display average with two digits of precision
            JOptionPane.showMessageDialog( null, "Class average is " + twoDigits.format( average ),

            // Class-average program with sentinel-controlled repetition.
import java.text.DecimalFormat;  // class to format numbers
import javax.swing.JOptionPane;

public class Average2 {
    public static void main( String args[] )
    {
        int total;           // sum of grades
        int gradeCounter;    // number of grades entered
        int grade;           // grade value

        double average;  // number with decimal point for average

        String gradeString;  // grade typed by user

        // initialization phase
        total = 0;         // initialize total
        gradeCounter = 0;  // initialize loop counter

        // processing phase
        // get first grade from user
        gradeString = JOptionPane.showInputDialog( "Enter Integer Grade or -1 to Quit:"

        // loop until sentinel value read from user
        while ( grade != -1 ) {
            total = total + grade;
            gradeCounter = gradeCounter + 1;

            // get next grade from user
            gradeString = JOptionPane.showInputDialog( "Enter Integer Grade or -1 to Quit:"

            // convert gradeString to int
            grade = Integer.parseInt( gradeString );

            // format numbers to nearest hundredth
        }

        // termination phase
        DecimalFormat twoDigits = new DecimalFormat( "0.00" );

        // if user entered at least one grade...
        if ( gradeCounter != 0 ) {
            // calculate average of all grades entered
            average = (double) total / gradeCounter;

            // display average with two digits of precision
            JOptionPane.showMessageDialog( null, "Class average is " + twoDigits.format( average ),

            // Class-average program with sentinel-controlled repetition.
import java.text.DecimalFormat;  // class to format numbers
import javax.swing.JOptionPane;
else // if no grades entered, output appropriate message
    JOptionPane.showMessageDialog( null, "No grades were entered", "Class Average", JOptionPane.INFORMATION_MESSAGE );

    System.exit( 0 ); // terminate application
}
} // end class Average2

4.10 Formulating Algorithms with Top-Down, Stepwise Refinement: Case Study 3
(Nested Control Structures)

- Nested control structures
Initialize passes to zero
Initialize failures to zero
Initialize student to one

While student counter is less than or equal to ten
    Input the next exam result
    If the student passed
        Add one to passes
    else
        Add one to failures
        Add one to student counter
    Print the number of passes
    Print the number of failures
    If more than eight students passed
        Print “Raise tuition”

Fig 4.10 Pseudocode for examination-results problem.
else // if result not 1, increment failures
    failures = failures + 1;

// increment studentCounter so loop eventually terminates
studentCounter = studentCounter + 1;
} // end while

// termination phase; prepare and display results
output = "Passed: " + passes + "Failed: " + failures;

// determine whether more than 8 students passed
if ( passes > 8 )
    output = output + "\nRaise Tuition";

JOptionPane.showMessageDialog( null, output,
    "Analysis of Examination Results",
    JOptionPane.INFORMATION_MESSAGE );

System.exit( 0 );  // terminate application
} // end main
} // end class Analysis

4.11 Compound Assignment Operators

• Assignment Operators
  – Abbreviate assignment expressions
  – Any statement of form
    • \texttt{variable = variable operator expression};
  – Can be written as
    • \texttt{variable operator = expression};
  – e.g., addition assignment operator +=
    • \texttt{c = c + 3}
    • \texttt{c += 3}
### 4.12 Increment and Decrement Operators

- **Unary increment operator (++)**
  - Increment variable’s value by 1
- **Unary decrement operator (--)**
  - Decrement variable’s value by 1
- **Preincrement / predecrement operator**
- **Post-increment / post-decrement operator**
<table>
<thead>
<tr>
<th>Operator</th>
<th>Called</th>
<th>Sample expression</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>++</td>
<td>preincrement</td>
<td>++a</td>
<td>Increment a by 1, then use the new value of a in the expression in which a resides.</td>
</tr>
<tr>
<td>++</td>
<td>postincrement</td>
<td>a++</td>
<td>Use the current value of a in the expression in which a resides, then increment a by 1.</td>
</tr>
<tr>
<td>--</td>
<td>predecrement</td>
<td>--b</td>
<td>Decrement b by 1, then use the new value of b in the expression in which b resides.</td>
</tr>
<tr>
<td>--</td>
<td>postdecrement</td>
<td>b--</td>
<td>Use the current value of b in the expression in which b resides, then decrement b by 1.</td>
</tr>
</tbody>
</table>

**Fig. 4.13** The increment and decrement operators.

```java
// Fig. 4.14: Increment.java
// Preincrementing and postincrementing operators.

public class Increment {

    public static void main( String args[] )
    {
        int c;

        // demonstrate postincrement
        c = 5;  // assign 5 to c
        System.out.println( c );  // print 5
        System.out.println( c++ );  // print 5 then postincrement
        System.out.println( c );  // print 6
        System.out.println();  // skip a line

        // demonstrate preincrement
        c = 5;  // assign 5 to c
        System.out.println( c );  // print 5
        System.out.println( ++c );  // preincrement then print 6
        System.out.println( c );  // print 6

    } // end main

} // end class Increment
```
### Operators

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>++</td>
<td>right to left</td>
<td>unary postfix</td>
</tr>
<tr>
<td>--</td>
<td>right to left</td>
<td>unary</td>
</tr>
<tr>
<td>+</td>
<td>left to right</td>
<td>multiplicative</td>
</tr>
<tr>
<td>-</td>
<td>left to right</td>
<td>additive</td>
</tr>
<tr>
<td>(type)</td>
<td>right to left</td>
<td>unary</td>
</tr>
<tr>
<td>&lt;</td>
<td>left to right</td>
<td>relational</td>
</tr>
<tr>
<td>&lt;=</td>
<td>left to right</td>
<td>equality</td>
</tr>
<tr>
<td>&gt;</td>
<td>left to right</td>
<td>conditional</td>
</tr>
<tr>
<td>&gt;=</td>
<td>left to right</td>
<td>assignment</td>
</tr>
<tr>
<td>==</td>
<td></td>
<td></td>
</tr>
<tr>
<td>!=</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Fig. 4.15** Precedence and associativity of the operators discussed so far.

---

### 4.13 Primitive Types

- **Primitive types**
  - “building blocks” for more complicated types

- **Java is strongly typed**
  - All variables in a Java program must have a type

- **Java primitive types**
  - Portable across computer platforms that support Java
### 5.2 Essentials of Counter-Controlled Repetition

- Counter-controlled repetition requires:
  - Control variable (loop counter)
  - Initial value of the control variable
  - Increment/decrement of control variable through each loop
  - Condition that tests for the final value of the control variable
5.3 for Repetition Statement

- Handles counter-controlled-repetition details
import java.awt.Graphics;
import javax.swing.JApplet;

public class ForCounter extends JApplet {

    // draw lines on applet's background
    public void paint( Graphics g )
    {
        super.paint( g );  // call paint method inherited from JApplet

        // for statement header includes initialization,
        // repetition condition and increment
        for ( int counter = 1; counter <= 10; counter++ )
            g.drawLine( 10, 10, 250, counter * 10 );

    } // end method paint

} // end class ForCounter

// Fig. 5.2: ForCounter.java
// Counter-controlled repetition with the for statement.

// Fig. 5.3 for statement header components.
5.3 for Repetition Structure (cont.)

for ( initialization; loopContinuationCondition; increment )
statement;

can usually be rewritten as:

initialization;
while ( loopContinuationCondition ){
    statement;
    increment;
}

Fig. 5.4 for statement activity diagram.
5.4 Examples Using the for Statement

- Varying control variable in for statement
  - Vary control variable from 1 to 100 in increments of 1
    • for (int i = 1; i <= 100; i++)
  - Vary control variable from 100 to 1 in increments of -1
    • for (int i = 100; i >= 1; i--)
  - Vary control variable from 7 to 77 in increments of 7
    • for (int i = 7; i <= 77; i += 7)

Outline

Sum.java

```java
// Fig. 5.5: Sum.java
// Summing integers with the for statement.
import javax.swing.JOptionPane;

public class Sum {
    public static void main(String args[]) {
        int total = 0; // initialize sum

        // total even integers from 2 through 100
        for (int number = 2; number <= 100; number += 2) total += number;

        // display results
        JOptionPane.showMessageDialog(null, "The sum is " + total,
                                    "Total Even Integers from 2 to 100",
                                    JOptionPane.INFORMATION_MESSAGE);

        System.exit(0); // terminate application
    }
}
```

Line 12

increment number by 2 each iteration
```java
// Fig. 5.6: Interest.java
// Calculating compound interest.
import java.text.NumberFormat;  // class for numeric formatting
import java.util.Locale;  // class for country-specific information
import javax.swing.JOptionPane;
import javax.swing.JTextArea;

public class Interest {

    public static void main( String args[] )
    {
        double amount;       // amount on deposit at end of each year
        double principal = 1000.0;  // initial amount before interest
        double rate = 0.05;         // interest rate

        NumberFormat moneyFormat = NumberFormat.getCurrencyInstance( Locale.US );

        JTextArea outputTextArea = new JTextArea();

        outputTextArea.setText( "Year	Amount on deposit
" );

        for ( int year = 1; year <= 10; year++ )
        {
            double amount = principal * Math.pow( 1.0 + rate, year );
            outputTextArea.append( year + "	" + moneyFormat.format( amount ) + "\n" );
        }

        JOptionPane.showMessageDialog( null, outputTextArea, "Compound Interest", JOptionPane.INFORMATION_MESSAGE );

        System.exit( 0 );  // terminate the application
    }
}
```

5.5 \textbf{do...while Repetition Statement}

- \textbf{do...while} structure
  - Similar to \textbf{while} structure
  - Tests loop-continuation after performing body of loop
    - i.e., loop body always executes at least once

Outline

DoWhileTest.java

\begin{verbatim}
// Fig. 5.7: DoWhileTest.java
// Using the do...while statement.
import java.awt.Graphics;
import javax.swing.JApplet;

public class DoWhileTest extends JApplet {

  public void paint( Graphics g ) {
    super.paint( g ); // call paint method inherited from JApplet
    int counter = 4; // initialize counter
    do {
      g.drawOval( 110 - counter * 10, 110 - counter * 10,
                  counter * 20, counter * 20 );
      ++counter;
    } while ( counter <= 10 ); // end do...while

    } // end method paint

} // end class DoWhileTest
\end{verbatim}

Oval is drawn before testing counter's final value
Fig. 5.8 do…while repetition statement activity diagram.

5.6 switch Multiple-Selection Statement

- switch statement
  - Used for multiple selections
Get user's input in JApplet

SwitchTest.java
Lines 16-21:
Getting user's input

switch (choice) { // determine shape to draw
    case 1: // draw a line
        g.drawLine(10, 10, 250, 10 + i * 10);
        break; // done processing case
    case 2: // draw a rectangle
        g.drawRect(10 + i * 10, 10 + i * 10,
                   50 + i * 10, 50 + i * 10);
        break; // done processing case
    case 3: // draw an oval
        g.drawOval(10 + i * 10, 10 + i * 10,
                   50 + i * 10, 50 + i * 10);
        break; // done processing case
    default: // draw string indicating invalid value entered
        g.drawString("Invalid value entered", 10, 20 + i * 15);
}
} // end switch
} // end for
} // end method paint
} // end class SwitchTest
Fig. 5.10 switch multiple-selection statement activity diagram with break statements.

5.7 break and continue Statements

- **break/continue**
  - Alter flow of control

- **break statement**
  - Causes immediate exit from control structure
    - Used in while, for, do...while or switch statements

- **continue statement**
  - Skips remaining statements in loop body
  - Proceeds to next iteration
    - Used in while, for or do...while statements
// Fig. 5.11: BreakTest.java
// Terminating a loop with break.
import javax.swing.JOptionPane;

public class BreakTest {
    public static void main( String args[] ) {
        String output = "";
        int count;

        for ( count = 1; count <= 10; count++ ) {  // loop 10 times
            if ( count == 5 )  // if count is 5,
                break;          // terminate loop
            output += count + " ";
        } // end for
        output += "Broke out of loop at count = "+ count;
        JOptionPane.showMessageDialog( null, output );
        System.exit( 0 );  // terminate application
    } // end main
} // end class BreakTest

// Fig. 5.12: ContinueTest.java
// Continuing with the next iteration of a loop.
import javax.swing.JOptionPane;

public class ContinueTest {
    public static void main( String args[] ) {
        String output = "";
        int count;

        for ( int count = 1; count <= 10; count++ ) {  // loop 10 times
            if ( count == 5 )  // if count is 5,
                continue;       // skip remaining code in loop
            output += count + " ";
        } // end for
        output += "Used continue to skip printing 5";
        JOptionPane.showMessageDialog( null, output );
        System.exit( 0 );  // terminate application
    } // end main
} // end class ContinueTest
5.9 Logical Operators

- Logical operators
  - Allows for forming more complex conditions
  - Combines simple conditions

- Java logical operators
  - & & (conditional AND)
  - & (boolean logical AND)
  - | | (conditional OR)
  - | (boolean logical inclusive OR)
  - ^ (boolean logical exclusive OR)
  - ! (logical NOT)

<table>
<thead>
<tr>
<th>expression1</th>
<th>expression2</th>
<th>expression1 &amp; &amp; expression2</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

*Fig. 5.15 & & (conditional AND) operator truth table.*

| expression1 | expression2 | expression1 | | expression2 |
|-------------|-------------|-------------|-----------------------------|
| false       | false       | false       | false                       |
| false       | true        | true        | true                        |
| true        | false       | true        | true                        |
| true        | true        | true        | true                        |

*Fig. 5.16 | | (conditional OR) operator truth table.*
expression1 | expression2 | expression1 ^ expression2
--- | --- | ---
false | false | false
false | true | true
true | false | true
true | true | false

Fig. 5.17 ∧ (boolean logical exclusive OR) operator truth table.

expression | !expression
--- | ---
false | true
true | false

Fig. 5.18 !(logical negation, or logical NOT) operator truth table.

```java
1      // Fig. 5.19: LogicalOperators.java
2      // Logical operators.
3      import javax.swing.*;
4
5      public class LogicalOperators
6
7      public static void main( String args[] )
8      {
9      // create JTextArea to display results
10     JTextArea outputArea = new JTextArea( 17, 20 );
11
12     // attach JTextArea to a JScrollPane so user can scroll results
13     JScrollPane scroller = new JScrollPane( outputArea );
14
15     // create truth table for && (conditional AND) operator
16     String output = "Logical AND (&&)
17     "false && false: " + ( false && false ) +
18     "false && true: " + ( false && true ) +
19     "true && false: " + ( true && false ) +
20     "true && true: " + ( true && true );
21
22     // create truth table for || (conditional OR) operator
23     output += "Logical OR (||)
24     "false || false: " + ( false || false ) +
25     "false || true: " + ( false || true ) +
26     "true || false: " + ( true || false ) +
27     "true || true: " + ( true || true );
28
// Fig. 5.19: Logical Operators
// Logical operators.
import javax.swing.*;

public class LogicalOperators

public static void main( String args[] )
{
    // create JTextArea to display results
    JTextArea outputArea = new JTextArea( 17, 20 );

    // attach JTextArea to a JScrollPane so user can scroll results
    JScrollPane scroller = new JScrollPane( outputArea );

    // create truth table for && (conditional AND) operator
    String output = "Logical AND (&&)
    "false && false: " + ( false && false ) +
    "false && true: " + ( false && true ) +
    "true && false: " + ( true && false ) +
    "true && true: " + ( true && true );

    // create truth table for || (conditional OR) operator
    output += "Logical OR (||)
    "false || false: " + ( false || false ) +
    "false || true: " + ( false || true ) +
    "true || false: " + ( true || false ) +
    "true || true: " + ( true || true );
```
// create truth table for & (boolean logical AND) operator
output += "\n\nBoolean logical AND (&)" +
"false & false: " + (false & false) +
"false & true: " + (false & true) +
"true & false: " + (true & false) +
"true & true: " + (true & true);

// create truth table for | (boolean logical inclusive OR) operator
output += "\n\nBoolean logical inclusive OR (|)
false | false: " + (false | false) +
"false | true: " + (false | true) +
"true | false: " + (true | false) +
"true | true: " + (true | true);

// create truth table for ^ (boolean logical exclusive OR) operator
output += "\n\nBoolean logical exclusive OR (^)
false ^ false: " + (false ^ false) +
"false ^ true: " + (false ^ true) +
"true ^ false: " + (true ^ false) +
"true ^ true: " + (true ^ true);

// create truth table for ! (logical negation) operator
output += "\n\nLogical NOT (!)
!false: " + (!false) +
"!true: " + (!true);

outputArea.setText(output);  // place results in JTextArea

JOptionPane.showMessageDialog( null, scroller,
    "Truth Tables", JOptionPane.INFORMATION_MESSAGE);

System.exit( 0 );  // terminate application

} // end main

} // end class LogicalOperators
### Operators

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>++  --</td>
<td>right to left</td>
<td>unary postfix</td>
</tr>
<tr>
<td>++  --  +  -  ! (type)</td>
<td>right to left</td>
<td>unary</td>
</tr>
<tr>
<td>*  /  %</td>
<td>left to right</td>
<td>multiplicative</td>
</tr>
<tr>
<td>+  -</td>
<td>left to right</td>
<td>additive</td>
</tr>
<tr>
<td>&lt;  &lt;=  &gt;  &gt;=</td>
<td>left to right</td>
<td>relational</td>
</tr>
<tr>
<td>==  !=</td>
<td>left to right</td>
<td>equality</td>
</tr>
<tr>
<td>&amp;</td>
<td>left to right</td>
<td>boolean logical AND</td>
</tr>
<tr>
<td>^</td>
<td>left to right</td>
<td>boolean logical OR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>left to right</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>left to right</td>
<td>conditional AND</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:=</td>
<td>right to left</td>
<td>conditional</td>
</tr>
<tr>
<td>+=  -=  *=  /=  %=</td>
<td>right to left</td>
<td>assignment</td>
</tr>
</tbody>
</table>

*Fig. 5.20* Precedence/associativity of the operators discussed so far.

---

### 5.10 Structured Programming Summary

- **Sequence structure**
  - “built-in” to Java
- **Selection structure**
  - *if, if...else* and *switch*
- **Repetition structure**
  - *while, do...while* and *for*
7.1 Introduction

• Arrays
  – Data structures
  – Related data items of same type
  – Can hold any data type or object
  – Remain same size once created
    •  *Fixed-length* entries

• Array
  – Group of variables
    • Have same type (primitive versus reference types)
  – Reference type (objects and arrays)
7.2 Arrays (cont.)

- Index
  - Also called subscript
  - Position number in square brackets
  - Must be positive integer or integer expression

  ```
  a = 5;
  b = 6;
  c[ a + b ] += 2;
  ```

  - Adds 2 to \( c[11] \)

- Using a value of type `long` as an index results in a compilation error.
7.2 Arrays (cont.)

- Examine array c
  - c is the array name
  - c.length accesses array c’s length (field of array object c)
  - c has 12 elements (c[0], c[1], ... c[11])
    - The value of c[0] is -45

- Difference between “i-th element of array” versus “array element i”

7.3 Declaring and Creating Arrays

- Declaring and Creating arrays
  - Arrays are objects that occupy memory
  - Created dynamically with keyword new
    ```
    int c[] = new int[12];
    ```
    - Equivalent to
      ```
      int c[]; // declare array variable
c = new int[12]; // create array
      ```
  - We can create arrays of objects too
    ```
    String b[] = new String[100];
    ```
  - int c[12]; // C/C++ syntax produces Java syntax error

- When creating arrays (default values for array elements):
  - Zero for numeric type elements
  - False for boolean elements
  - Null for references (any non-primitive type)
7.4 Examples Using Arrays

• Declaring arrays
• Creating arrays
• Initializing arrays
• Manipulating array elements
7.4 Examples Using Arrays (Cont.)

- Using an array initializer
  - Use *initializer list*
    - Items enclosed in braces ({})
    - Items in list separated by commas
      ```java
      int n[] = { 10, 20, 30, 40, 50 };  
      ```
      - Creates a five-element array
      - Index values of 0, 1, 2, 3, 4
    - Do not need keyword `new`
```
1 // Fig. 7.3: InitArray.java
2 // Initializing an array with a declaration.
3 import javax.swing.*;
4 public class InitArray {
5     public static void main( String args[] )
6     {
7         int[] array = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
8         String output = "Index\tValue\n";
9         for ( int counter = 0; counter < array.length; counter++ )
10             output += counter + "\t" + array[ counter ] + "\n";
11     } // end main
12 } // end class InitArray
```

Outline

InitArray.java

Line 11
Declare array as an array of ints

Compiler uses initializer list to allocate array

Each array element corresponds to element in initializer list

Line 11
Compile uses initializer list to allocate array
7.4 Examples Using Arrays (Cont.)

• Calculating the value to store in each array element
  – Initialize elements of 10-element array to even integers

```
// Fig. 7.4: InitArray.java
// Initialize array with the even integers
import javax.swing.*;

public class InitArray {
    public static void main( String args[] ) {
        final int ARRAY_LENGTH = 10;    // constant
        int array[];                    // reference to int array
        array = new int[ ARRAY_LENGTH ];  // create array

        // calculate value for each array element
        for ( int counter = 0; counter < array.length; counter++ )
            array[ counter ] = 2 + 2 * counter;

        String output = "Index	Value
```
```
7.4 Examples Using Arrays (Cont.)

- Summing the elements of an array
  - Array elements can represent a series of values
    - We can sum these values
7.4 Examples Using Arrays (Cont.)

- Using histograms do display array data graphically
  - Histogram
    - Plot each numeric value as bar of asterisks (*)
```
// Fig. 7.6: Histogram.java
// Histogram printing program.
import javax.swing.*;

public class Histogram {
    public static void main( String args[] )
    {
        int array[] = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };
        String output = "Element\tValue\tHistogram";
        // for each array element, output a bar in histogram
        for ( int counter = 0; counter < array.length; counter++ ) {
            output += "\n" + counter + "\t" + array[ counter ] + "\t";
            // print bar of asterisks
            for ( int stars = 0; stars < array[ counter ]; stars++ )
                output += "*";
        } // end outer for
        JTextArea outputArea = new JTextArea();
        outputArea.setText( output );
        JOptionPane.showMessageDialog( null, outputArea,
            "Histogram Printing Program", JOptionPane.INFORMATION_MESSAGE );
        System.exit( 0 );
    } // end main
} // end class Histogram
```

Outline

Histogram.java

Line 9
Declare array with initializer list

Line 19
For each array element, print associated number of asterisks
7.4 Examples Using Arrays (Cont.)

• Using the elements of an array as counters
  – Use a series of counter variables to summarize data

```java
// Fig. 7.7: RollDie.java
// Roll a six-sided die 6000 times.
import javax.swing.*;

public class RollDie {

    public static void main( String args[] )
    {        
        int frequency[] = new int[7];
        
        // roll die 6000 times; use die value as frequency index
        for ( int roll = 1; roll <= 6000; roll++ )
            ++frequency[ 1 + ( int ) ( Math.random() * 6 ) ];

        String output = "Face	Frequency"
        
        // append frequencies to String output
        for ( int face = 1; face < frequency.length; face++ )
            output += "n" + face + "\t" + frequency[ face ];

        JTextArea outputArea = new JTextArea();
        outputArea .setText( output );
        
        JOptionPane .showMessageDialog ( null, outputArea,
            "Rolling a Die 6000 Times" , JOptionPane .INFORMATION_MESSAGE );

        System.exit( 0 );
    }
}
```

Outline

- Declare frequency as array of 7 ints
- Generate 6000 random integers in range 1-6
- Increment frequency values at index associated with random number
7.4 Examples Using Arrays (Cont.)

- Using arrays to analyze survey results
  - 40 students rate the quality of food
    - 1-10 Rating scale: 1 mean awful, 10 means excellent
  - Place 40 responses in array of integers
  - Summarize results

```java
// Fig. 7.8: StudentPoll.java
// Student poll program.
import javax.swing.*;

public class StudentPoll {
    public static void main( String args[] )
    {
        int responses[] = { 1, 2, 6, 4, 8, 5, 9, 7, 8, 10, 1, 6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 8, 6, 7, 5, 6, 7, 5, 6, 8, 6, 7, 5, 6, 7, 5, 6, 4, 8, 6, 8, 10 }; // for each answer, select responses element and use that value as frequency index to determine element to increment
        int frequency[] = new int[11]; // for each answer, select responses element and use that value as frequency index to determine element to increment
        for ( int answer = 0; answer < responses.length; answer++ )
            ++frequency[ responses[ answer ] ]; // for each answer, select responses element and use that value as frequency index to determine element to increment

        String output = "Rating	Frequency
";// append frequencies to String output
        for ( int rating = 1; rating < frequency.length; rating++ )
            output += rating + "	" + frequency[ rating ] + "\n"; // append frequencies to String output
        JTextArea outputArea = new JTextArea();
        outputArea.setText( output );
        outputArea.setLineWrap( false );
        JOptionPane.showMessageDialog( null, outputArea );
    }
}
```

Outline

- Declare `responses` as array to store 40 responses
  - Lines 9-11
- Declare `frequency` as array of 11 `int`, and ignore the first element
  - Line 12
- For each response, increment `frequency` values at index associated with that response
  - Lines 16-17
7.4 Examples Using Arrays (Cont.)

- Some additional points
  - When looping through an array
    - Index should never go below 0
    - Index should be less than total number of array elements
  - When invalid array reference occurs (0 > index or index >= length)
    - Java generates ArrayIndexOutOfBoundsException
  - Chapter 15 discusses exception handling
7.5 References and Reference Parameters

- 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
    - Changes of arguments in method do not affect value of caller.
  - 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
  - Java does not allow programmers to choose whether to pass
    an argument by pass-by-value or by pass-by-reference.

7.6 Passing Arrays to Methods and Return Information

- To pass array argument to a method
  - Specify array name without brackets
    - Array `hourlyTemperatures` is declared as
      ```java
      int hourlyTemperatures = new int[24];
      ```
    - The method call
      ```java
      modifyArray(hourlyTemperatures);
      ```
    - Passes array `hourlyTemperatures` to method `modifyArray`
- Return information of a method:
  - Primitive types are returned by value
  - Objects are returned by reference
import java.awt.Container;
import javax.swing.*;

public class PassArray extends JApplet {
  public void init()
  {
    JTextArea outputArea = new JTextArea();
    Container container = getContentPane();
    container.add(outputArea);

    int array[] = { 1, 2, 3, 4, 5 };
    String output = "Effects of passing entire array by reference: 
    " + "The values of the original array are: 
    
    for ( int counter = 0; counter < array.length; counter++ )
    output += "   " + array[counter];

    modifyArray( array ); // array passed by reference

    output += 
    "The values of the modified array are: 
    
    for ( int counter = 0; counter < array.length; counter++ )
    output += "   " + array[counter];

    output += "Effects of passing array element by value: 
    " + "array[3] before modifyElement : " + array[3];
    modifyElement( array[3] ); // attempt to modify array[3]
    output += "array[3] after modifyElement : " + array[3];

    outputArea.setText(output);
  }

  public void modifyArray( int array2[] )
  {
    for ( int counter = 0; counter < array2.length; counter++ )
    array2[counter] *= 2;
  }

  public void modifyElement( int element )
  {
    element *= 2;
  }
}

} // end class PassArray
7.7 Sorting Arrays

• Sorting data
  – Attracted intense research in computer-science field
  – Bubble sort
    • Smaller values “bubble” their way to top of array
    • Larger values “sink” to bottom of array
    • Use nested loops to make several passes through array
      – Each pass compares successive pairs of elements
        • Pairs are left along if increasing order (or equal)
        • Pairs are swapped if decreasing order
// Fig. 7.10: BubbleSort.java
// Sort an array’s values into ascending order.
import java.awt.*;
import javax.swing.*;

public class BubbleSort extends JApplet {
    public void init() {
        JTextArea outputArea = new JTextArea();
        Container container = getContentPane();
        container.add(outputArea);
        int array[] = {2, 6, 4, 8, 10, 12, 89, 68, 45, 37};
        String output = "Data items in original order
        " + "; // append original array values to String output
        for (int counter = 0; counter < array.length; counter++)
            output += "   " + array[counter];
        bubbleSort(array); // sort array
        output += "Data items in ascending order
        " + "; // append sorted array values to String output
        for (int counter = 0; counter < array.length; counter++)
            output += "   " + array[counter];
        outputArea.setText(output);
    } // end method init

    public void bubbleSort(int array2[]) {
        for (int pass = 1; pass < array2.length; pass++) {
            for (int element = 0; element < array2.length - 1; element++) {
                // compare side-by-side elements and swap them if
                // first element is greater than second element
                if (array2[element] > array2[element + 1])
                    swap(array2, element, element + 1);
            } // end loop to control comparisons
        } // end loop to control passes
    } // end method bubbleSort

7.8 Searching Arrays: Linear Search and Binary Search

- Searching
  - Finding elements in large amounts of data
    - Determine whether array contains value matching key value
  - Linear searching
  - Binary searching
7.8 Searching Arrays: Linear Search and Binary Search (Cont.)

• Linear search
  – Compare each array element with search key
    • If search key found, return element index
    • If search key not found, return \(-1\) (invalid index)
  – Works best for small or unsorted arrays
  – Inefficient for larger arrays

```
// Fig. 7.11: LinearSearch.java
// Linear search of an array.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class LinearSearch extends JApplet implements ActionListener {

    // set up applet's GUI
    public void init()
    {
        // get content pane and set its layout to FlowLayout
        Container container = getContentPane();
        container.setLayout( new FlowLayout() );

        // set up JLabel and JTextField for user input
        JLabel enterLabel, resultLabel;
        JTextField enterField, resultField;
        int array[];

        // register this applet as enterField's action listener
        enterField.addActionListener( this );
    }
```

Outline

LinearSearch.java

Line 11
Declare array of ints
linearSearch.java

Lines 39-42
Allocate 100 ints for array and populate array with even ints

Line 50
Loop through array

Lines 53-54
If array element at index matches search key, return index

Invoked when user presses Enter

Invoke method linearSearch, using array and search key as arguments

Invoked when user enters data

Invoke method linearSearch, using array and search key as arguments
7.8 Searching Arrays: Linear Search and Binary Search (Cont.)

• Binary search
  – Efficient for large, sorted arrays
  – Eliminates half of the elements in search through each pass
    • Compare middle array element to search key
      – If element equals key
        • Return array index
      – If element is less than key
        • Repeat search on first half of array
      – If element is greater than key
        • Repeat search on second half of array
    – Continue search until
      • element equals search key (success)
      • Search contains one element not equal to key (failure)
// register this applet as enterField's action listener
enterField.addActionListener(this);

// set up JLabel and JTextField for displaying results
resultLabel = new JLabel("Result");
container.add(resultLabel);

resultField = new JTextField(20);
resultField.setEditable(false);
container.add(resultField);

// create array and fill with even integers 0 to 28
array = new int[15];
for (int counter = 0; counter < array.length; counter++)
array[counter] = 2 * counter;

// obtain user input and call method binarySearch
public void actionPerformed(ActionEvent actionEvent) {
    String searchKey = actionEvent.getActionCommand();
    int element = binarySearch(array, Integer.parseInt(searchKey));
    output.setText(display);
    if (element != -1)
        resultField.setText("Found value in element "+element);
    else
        resultField.setText("Value not found");
}

// method to perform binary search of an array
public int binarySearch(int array2[], int key) {
    int low = 0;  // low element index
    int high = array2.length - 1;  // high element index
    int middle;  // middle element index
    while (low <= high) {
        middle = (low + high) / 2;  // determine middle element index
        buildOutput(array2, low, middle, high);
        if (key == array2[middle])
            return middle;
        else if (key < array2[middle])
            high = middle - 1;
        else
            low = middle + 1;
    }
    return -1;
}
If search key matches middle array element, return element index

Lines 93-94
If search key is less than middle array element, repeat search on first array half

Lines 97-98
If search key is greater than middle array element, repeat search on second array half

Lines 101-102
Method buildOutput displays array contents being searched

Display an asterisk next to middle element

Lines 128
Display an asterisk next to middle element

Method buildOutput displays array contents being searched
7.9 Multidimensional Arrays

- Multidimensional arrays
  - Tables with rows and columns
    - Two-dimensional array
      - Declaring two-dimensional array b[2][2]
        ```java
        int b[][] = { { 1, 2 }, { 3, 4 } };
        ```
      - 1 and 2 initialize b[0][0] and b[0][1]
      - 3 and 4 initialize b[1][0] and b[1][1]
        ```java
        int b[][] = { { 1, 2 }, { 3, 4, 5 } };
        ```
      - row 0 contains elements 1 and 2
      - row 1 contains elements 3, 4 and 5
7.9 Multidimensional Arrays (Cont.)

- Creating multidimensional arrays
  - Can be allocated dynamically
    - 3-by-4 array
      ```java
      int b[][];
      b = new int[3][4];
      ```
    - Rows can have different number of columns
      ```java
      int b[][];
      b = new int[2][]; // allocate rows
      b[0] = new int[5]; // allocate row 0
      b[1] = new int[3]; // allocate row 1
      ```

Fig. 7.13 Two-dimensional array with three rows and four columns.
// Fig. 7.14: InitArray.java
// Initializing two-dimensional arrays.
import java.awt.Container;
import javax.swing.*;

public class InitArray extends JApplet {

    JTextArea outputArea;

    // set up GUI and initialize applet
    public void init() {
        outputArea = new JTextArea();
        Container container = getContentPane();
        container.add( outputArea );

        int array1[][] = { { 1, 2, 3 }, { 4, 5, 6 } };
        int array2[][] = { { 1, 2 }, { 3 }, { 4, 5, 6 } };

        outputArea.setText( "Values in array1 by row are
" );
        buildOutput( array1 );

        outputArea.append( "\nValues in array2 by row are\n" );
        buildOutput( array2 );
    }

    // append rows and columns of an array to outputArea
    public void buildOutput ( int array[][] ) {
        // loop through array's rows
        for ( int row = 0; row < array.length; row++ ) {
            // loop through columns of current row
            for ( int column = 0; column < array[ row ].length; column++ ) {
                outputArea.append( array[ row ][ column ] + "  " );
            }
            outputArea.append( "\n" );
        }
    }

} // end class InitArray

array[row].length returns number of columns associated with row subscript
array[row].length returns number of columns associated with row subscript
// Fig. 7.15: DoubleArray.java
// Two-dimensional array example.
import java.awt.*;
import javax.swing.*;

public class DoubleArray extends JApplet {

int grades[][] = { { 77, 68, 86, 73 },
{ 96, 87, 89, 81 },
{ 70, 90, 86, 81 } };

int students, exams;
String output;
JTextArea outputArea;

// initialize fields
public void init()
{
students = grades.length;     // number of students
exams = grades[ 0 ].length;   // number of exams

// create JTextArea and attach to applet
outputArea = new JTextArea();
Container container = getContentPane();
container.add( outputArea );

// build output string
output = "The array is: 
";
buildString();

// call methods minimum and maximum
output += "

Lowest grade: " + minimum() + "
Highest grade: " + maximum() + "
";

// call method average to calculate each student's average
for ( int counter = 0; counter < students; counter++ )
output += "Average for student " + counter + " is " +
average( grades[ counter ] ); // pass one row of array grades

// change outputArea's display font
outputArea.setFont( new Font( "Monospaced", Font.PLAIN, 12 ) );

// place output string in outputArea
outputArea.setText( output );
}

// find minimum grade
public int minimum()
{
int lowGrade = grades[ 0 ][ 0 ];

// assume first element of grades array is smallest
int lowGrade = grades[ 0 ][ 0 ];
}
Outline

DoubleArray.java

Lines 54-61
Use a nested loop to search for lowest grade in series

Lines 74-81
Use a nested loop to search for highest grade in series

Method **average** takes array of student test results as parameter

Calculate sum of array elements

Divide by number of elements to get average

Method **average** takes array of student test results as parameter

Calculate sum of array elements

Divide by number of elements to get average
Summary

- Array is an indexed collection of data values
- Array elements: values of primitive data type or objects
- All array elements must have same data type
- Array is created with new operator
- Array can have multiple indices.
- Arrays are passed to methods by reference
Part 8: Collections

- Arrays
- Collections
- Lists
- Sets
- Maps

22.2 Collections Overview

- Collection
  - Data structure (object) that can hold references to other objects
- Collection Interfaces
  - Interfaces declare operations for various collection types
  - Belong to package java.util
    - Collection
    - Set
    - List
    - Map
- Collection implementations
  - execute these operations
- Collections framework
  - contains several implementations of Collection Interfaces
22.3 Class Arrays

- Class Arrays
  - Provides static methods for manipulating arrays
  - Provides “high-level” methods
    - Method `binarySearch` for searching sorted arrays
    - Method `equals` for comparing arrays
    - Method `fill` for placing values into arrays
    - Method `sort` for sorting arrays

```java
// Fig. 22.1: UsingArrays.java
// Using Java arrays.
import java.util.*;

public class UsingArrays {
    private int intValues[] = { 1, 2, 3, 4, 5, 6 };
    private double doubleValues[] = { 8.4, 9.3, 0.2, 7.9, 3.4 };
    private int filledInt[], intValuesCopy[];

    // initialize arrays
    public UsingArrays() {
        filledInt = new int[10];
        intValuesCopy = new int[intValues.length];
        Arrays.fill(filledInt, 7);   // fill with 7s
        Arrays.sort(doubleValues);   // sort doubleValues ascending
        // copy array intValues into array intValuesCopy
        System.arraycopy(intValues, 0, intValuesCopy, 0, intValues.length);
    }
}
```

Use static method `fill` of class `Arrays` to populate array with 7s.

Use static method `sort` of class `Arrays` to sort array’s elements in ascending order.

Use static method `arraycopy` of class `System` to copy array `intValues` into array `intValuesCopy`. 
// output values in each array
public void printArrays()
{
    System.out.print( "doubleValues: ");
    for ( int count = 0; count < doubleValues.length; count++ )
        System.out.print( doubleValues[ count ] + " ");
    System.out.print( "\n"
    for ( int count = 0; count < intValues.length; count++ )
        System.out.print( intValues[ count ] + " ");
    System.out.print( "\n"
    for ( int count = 0; count < filledInt.length; count++ )
        System.out.print( filledInt[ count ] + " ");
    System.out.print( "\n"
    for ( int count = 0; count < intValuesCopy.length; count++ )
        System.out.print( intValuesCopy[ count ] + " ");
    System.out.println();
} // end method printArrays

// find value in array intValues
public int searchForInt( int value )
{
    return Arrays.binarySearch( intValues, value );
}

// compare array contents
public void printEquality()
{
    boolean b = Arrays.equals( intValues, intValuesCopy );
    System.out.println( "intValues " + ( b ? "==" : "!=" ) + " intValuesCopy" );
    b = Arrays.equals( intValues, filledInt );
    System.out.println( "intValues " + ( b ? "==" : "!=" ) + " filledInt" );
}
public static void main( String args[] )
{
    UsingArrays usingArrays = new UsingArrays();
    usingArrays.printArrays();
    usingArrays.printEquality();
    int location = usingArrays.searchForInt( 5 );
    System.out.println( ( location >= 0 ? "Found 5 at element " +
                        location : "5 not found" ) + " in intValues" );
    location = usingArrays.searchForInt( 8763 );
    System.out.println( ( location >= 0 ? "Found 8763 at element " +
                        location : "8763 not found" ) + " in intValues" );
}
} // end class UsingArrays

doubleValues: 0.2 3.4 7.9 8.4 9.3
intValues: 1 2 3 4 5 6
filledInt: 7 7 7 7 7 7
intValuesCopy: 1 2 3 4 5 6
intValues != filledInt

Found 5 at element 4 in intValues
8763 not found in intValues

// Fig. 22.2: UsingAsList.java
// Using method asList.
import java.util.*;

public class UsingAsList {
    private static final String values[] = { "red", "white", "blue" };  
    private List list;

    // initialize List and set value at location 1
    public UsingAsList() {
        List = Arrays.asList( values );   // get List
        list.set( 1, "green" );           // change a value
    }

    // output List and array
    public void printElements() {
        System.out.print( "List elements : " );
        for ( int count = 0; count < list.size(); count++ )
            System.out.print( list.get( count ) + "");
        System.out.print( "\nArray elements: " );
    }
}

Collections Framework enables the programmer to view elements of
one collection type as elements of a different collection type.

E.g. Arrays can be viewed as Lists
• Arrays.asList creates a fixed-sized List that operates faster than any provided List implementations
• A List created with Arrays.asList has fixed size.
  • calling methods add or remove throws an UnsupportedOperationException

### 22.4 Interface Collection and Class Collections

- **Interface Collection**
  - root interface in the Collections hierarchy
  - Contains *bulk operations*
    - Adding, clearing, comparing and retaining objects
  - Interfaces `Set` and `List` extend interface `Collection`

- **Class collections**
  - Provides *static* methods that manipulate collections
  - Collections can be manipulated polymorphically
22.5 Lists

- List
  - Ordered Collection that can contain duplicate elements
  - Sometimes called a sequence
  - 1st List element index is 0.
  - Several classes implement interface List
    - **ArrayList**: resizable-array implementation of a List
    - **LinkedList**: linked-list implementation of a list
    - **Vector**: similar to ArrayList but worse performance

---

```java
// Fig. 22.3: CollectionTest.java
// Using the Collection interface.
import java.awt.Color;
import java.util.*;

public class CollectionTest {
    private static final String colors[] = { "red", "white", "blue" };

    // create ArrayList, add objects to it and manipulate it
    public CollectionTest() {
        List list = new ArrayList();

        // add objects to list
        list.add( Color.MAGENTA ); // add a color object

        for ( int count = 0; count < colors.length; count++ )
            list.add( colors[ count ] );

        list.add( Color.CYAN ); // add a color object

        // output list contents
        System.out.println( "ArrayList: " );

        for ( int count = 0; count < list.size(); count++ )
            System.out.print( list.get( count ) + " " );
    }
}
```

- Use `List` method `add` to add objects to `ArrayList`
- `List` method `get` returns individual element in `List`
Method `removeStrings` takes a `Collection` as an argument, Line 31 passes `List`, which extends `Collection`, to this method.

`Iterator` is more flexible than `Enumeration`.
`Iterator` can remove elements from collection.
`Enumeration` cannot remove elements from collections.

```
28    // remove all String objects
29    removeStrings( list );
30
31    // output list contents
32    System.out.println( "\n\nArrayList after calling removeStrings: \n\n" );
33    for ( int count = 0; count < list.size(); count++ )
34        System.out.print( list.get( count ) + " " );
35
36    } // end constructor CollectionTest
37
38    // remove String objects from Collection
39    private void removeStrings( Collection collection )
40    {
41        Iterator iterator = collection.iterator(); // get iterator
42
43        // loop while collection has items
44        while ( iterator.hasNext() )
45            if ( iterator.next() instanceof String )
46                iterator.remove();  // remove String object
47
48    } // end method removeStrings
```
// Fig. 22.4: ListTest.java
// Using LinkLists.
import java.util.*;

public class ListTest {
    private static final String colors[] = { "black", "yellow", "green", "blue", "violet", "silver" };
    private static final String colors2[] = { "gold", "white", "brown", "blue", "gray", "silver" };

    // set up and manipulate LinkedList objects
    public ListTest() {
        List link = new LinkedList();
        List link2 = new LinkedList();

        // add elements to each list
        for ( int count = 0; count < colors.length; count++ ) {
            link.add( colors[ count ] );
            link2.add( colors2[ count ] );
        }

        link.addAll( link2 ); // concatenate lists
        link2 = null; // release resources

        printList( link );
        uppercaseStrings( link );
        printList( link );

        System.out.print( "Deleting elements 4 to 6..." );
        removeItems( link, 4, 7 );
        printList( link );
        printReversedList( link );
    } // end constructor ListTest

    // output List contents
    public void printList( List list ) {
        System.out.println( "list: " );
        for ( int count = 0; count < list.size(); count++ )
            System.out.print( list.get( count ) + " " );
        System.out.println();
    }

    // utility methods
    public void uppercaseStrings( List link ) {
    }

    public void printReversedList( List link ) {
    }

    public void removeItems( List link, int start, int num ) {
    }

    // end class ListTest
}
private void uppercaseStrings(List list) {
    ListIterator iterator = list.listIterator();
    while (iterator.hasNext()) {
        Object object = iterator.next(); // get item
        if (object instanceof String) { // check for String
            iterator.set(((String) object).toUpperCase());
        }
    }
}

private void removeItems(List list, int start, int end) {
    list.subList(start, end).clear(); // remove items
}

private void printReversedList(List list) {
    ListIterator iterator = list.listIterator(list.size());
    
    Use ListIterator to traverse LinkedList elements and convert them to
    upper case (if elements are Strings)
    
    Use List methods subList and clear
to remove LinkedList elements

ListTest.java
Line 79
Line 80
76    System.out.println( "Reversed List:" );
77    
78    // print list in reverse order
79    while( iterator.hasPrevious() )
70      System.out.print( iterator.previous() + " " );
81    }
82    
83    public static void main( String args[] )
84    {
85      new ListTest();
86    }
87    
88 } // end class ListTest

Deleting elements 4 to 6...
list:
BLACK YELLOW GREEN BLUE WHITE BROWN BLUE GRAY SILVER

Reversed List:
SILVER GRAY BLUE BROWN WHITE BLUE GREEN YELLOW BLACK

UsingToArray.java
Line 20
1      // Fig. 22.5: UsingToArray.java
2      // Using method toArray.
3      import java.util.*;
4      
5      public class UsingToArray {
6      // create LinkedList, add elements and convert to array
7      public UsingToArray() {
8      
9        String colors[] = { "black", "blue", "yellow" };
10       LinkedList links = new LinkedList( Arrays.asList( colors ) );
11       
12       links.addLast( "red" ); // add as last item
13       links.add( "pink" ); // add to the end
14       links.add( 3, "green" ); // add at 3rd index
15       links.addFirst( "cyan" ); // add as first item
16       
17       // get LinkedList elements as an array
18       colors = ( String[] ) links.toArray( new String[ links.size() ] );
19       
20       System.out.println( "colors: " );
21       
22 } // end class UsingToArray
22.6 Algorithms

- Collections Framework provides set of algorithms
  - Implemented as static methods
    - algorithms of class List
      - sort
      - binarySearch
      - reverse
      - shuffle
      - fill
      - copy
    - algorithms of class Collections
      - min
      - max
- The collections-framework algorithms are polymorphic.
22.6.1 Algorithm sort

• sort
  – Sorts List elements
    • Order is determined by natural order of elements’ type
    • Relatively fast: n log(n)
  – separate parameter can be used to modify ordering of elements

```java
// Fig. 22.6: Sort1.java
// Using algorithm sort.
import java.util.*;

public class Sort1 {
    private static final String suits[] = {
        "Hearts", "Diamonds", "Clubs", "Spades" };

    public void printElements() {
        // create ArrayList
        List list = new ArrayList(arrays.asList(suits));

        // output list
        System.out.println( "Unsorted array elements:\n" + list );

        Collections.sort(list); // sort ArrayList

        // output list
        System.out.println( "Sorted array elements:\n" + list );
    }
}
```
Unsorted array elements:
[Hearts, Diamonds, Clubs, Spades]
Sorted array elements:
[Clubs, Diamonds, Hearts, Spades]

```java
public static void main(String args[])
{
    new Sort1().printElements();
}
// end class Sort1
```

```
// Fig. 22.7: Sort2.java
// Using a Comparator object with algorithm sort.
import java.util.*;

public class Sort2 {
    private static final String suits[] =
        { "Hearts", "Diamonds", "Clubs", "Spades" };

    // output List elements
    public void printElements()
    {
        List list = Arrays.asList(suits); // create List

        // output List elements
        System.out.println("Unsorted array elements:
" + list);

        // sort in descending order using a comparator
        Collections.sort(list, Collections.reverseOrder());

        // output List elements
        System.out.println("Sorted list elements:
" + list);
    }
}
```

**Method `reverseOrder` of class `Collections` returns a `Comparator` object that represents the collection’s reverse order.**

**Method `sort` of class `Collections` can use a `Comparator` object to sort a `List`**
public static void main( String args[] )
{
    new Sort2().printElements();
}

Sort in order using a custom comparator TimeComparator.
public static void main( String args[] ) {
    new Sort2().printElements();
}

private class TimeComparator implements Comparator {
    int hourCompare, minuteCompare, secondCompare;
    Time2 time1, time2;
    
    public int compare(Object object1, Object object2) {
        // cast the objects
        time1 = (Time2)object1;
        time2 = (Time2)object2;
        hourCompare = new Integer( time1.getHour() ).compareTo(
            new Integer( time2.getHour() ) );
        // test the hour first
        if ( hourCompare != 0 )
            return hourCompare;
        minuteCompare = new Integer( time1.getMinute() ).compareTo(
            new Integer( time2.getMinute() ) );
        // then test the minute
        if ( minuteCompare != 0 )
            return minuteCompare;
        secondCompare = new Integer( time1.getSecond() ).compareTo(
            new Integer( time2.getSecond() ) );
        return secondCompare; // return result of comparing seconds
    }
} // end class TimeComparator

} // end class Sort3

Unsorted array elements:
[06:24:34, 18:14:05, 08:05:00, 12:07:58, 06:14:22]
Sorted list elements:
[06:14:22, 06:24:34, 08:05:00, 12:07:58, 18:14:05]
22.6.2 Algorithm shuffle

- shuffle
  - Randomly orders List elements

```java
// Fig. 22.9: Cards.java
// Using algorithm shuffle.
import java.util.*;

// class to represent a Card in a deck of cards
class Card {
    private String face;
    private String suit;

    // initialize a Card
    public Card( String initialface, String initialSuit )
    {
        face = initialface;
        suit = initialSuit;
    }

    // return face of Card
    public String getFace()
    {
        return face;
    }

    // return suit of Card
    public String getSuit()
    {
        return suit;
    }
}
```
public String toString()
{
    StringBuffer buffer = new StringBuffer( face + " of " + suit);
    buffer.setLength(20);
    return buffer.toString();
}

// class Cards declaration
public class Cards
{
    private static final String suits[] = {
        "Hearts", "Clubs", "Diamonds", "Spades" };
    private static final String faces[] = {
        "Ace", "Deuce", "Three", 
        "Jack", "Queen", "King" };
    private List list;

    // set up deck of Cards and shuffle
    public Cards()
    {
        Card deck[] = new Card[52];
        for ( int count = 0; count < deck.length; count++ )
            deck[ count ] = new Card( faces[ count % 13 ],
            suits[ count / 13 ] );
        list = Arrays.asList( deck );   // get List
        Collections.shuffle( list );    // shuffle deck
    }

    // output deck
    public void printCards()
    {
        int half = list.size() / 2 - 1;
        for ( int i = 0, j = half + 1; i <= half; i++, j++ )
            System.out.println( list.get( i ).toString() + list.get( j ).toString());
    }

    public static void main( String args[] )
    {
        new Cards().printCards();
    }
} // end class Cards
### 22.6.3 Algorithms reverse, fill, copy, max and min

Class Collections provides algorithms:

- **reverse**
  - Reverses the order of List elements
- **fill**
  - Populates List elements with values
- **copy**
  - Creates copy of a List
- **max**
  - Returns largest element in List
- **min**
  - Returns smallest element in List
import java.util.

public class Algorithms1 {
    private String letters[] = { "P", "C", "M" }, lettersCopy[];
    private List list, copyList;
    public Algorithms1()
    {
        list = Arrays.asList( letters ); // get List
        lettersCopy = new String[3];
        copyList = Arrays.asList( lettersCopy );
        System.out.println( "Initial list: " );
        output( list );
        Collections.reverse( list ); // reverse order
        System.out.println( "
After calling reverse: " );
        output( list );
        Collections.copy( copyList, list ); // copy List
        System.out.println( "
After copying: " );
        output( copyList );
        Collections.fill( list, "R" ); // fill list with Rs
        System.out.println( "
After calling fill: " );
        output( list );
    } // end constructor
    public void output( List listRef )
    {
        System.out.print( "The list is: " );
        System.out.print( "
Max: " + Collections.max( listRef ) );
        System.out.println( "  Min: " + Collections.min( listRef ) );
    }
    public static void main( String args[] )
    {
        new Algorithms1();
    } // end class Algorithms1
22.6.4 Algorithm binarySearch

- binarySearch
  - static method of class Collections
  - Locates Object in List (i.e. LinkedList, Vector, ArrayList)
    - Returns index of Object in List if Object exists
    - Returns negative value if Object does not exist
    - If multiple list elements match with search key, no guarantee which one will be returned
  - prerequisite: list’s element must be sorted in ascending order.
public class BinarySearchTest {
    private static final String colors[] = { "red", "white", "blue", "black", "yellow", "purple", "tan", "pink" };
    private List list; // List reference

    public BinarySearchTest() {
        list = new ArrayList( Arrays.asList( colors ) );
        Collections.sort( list ); // sort the ArrayList
        System.out.println( "Sorted ArrayList: " + list );
    }

    public void printSearchResults() {
        printSearchResultsHelper( colors[ 3 ] ); // first item
        printSearchResultsHelper( colors[ 0 ] ); // middle item
        printSearchResultsHelper( colors[ 7 ] ); // last item
        printSearchResultsHelper( "aardvark" ); // below lowest
        printSearchResultsHelper( "goat" ); // does not exist
        printSearchResultsHelper( "zebra" ); // does not exist
    }

    private void printSearchResultsHelper( String key ) {
        int result = 0;
        System.out.println( "Searching for: " + key );
        result = Collections.binarySearch( list, key );
        System.out.println( ( result >= 0 ? "Found at index " + result : "Not Found (" + result + ")" ) );
    }

    public static void main( String args[] ) {
        new BinarySearchTest().printSearchResults();
    }
} // end class BinarySearchTest

Sorted ArrayList: black blue pink purple red tan white yellow
Searching for: black
   Found at index 0
Searching for: red
   Found at index 4
Searching for: pink
   Found at index 2
Searching for: aardvark
         Not Found (-1)
Searching for: goat
         Not Found (-3)
Searching for: zebra
         Not Found (-9)
22.7 Sets

- **Set**
  - Collection that contains unique elements
  - Collections framework contains several Set implementations:
    - **HashSet**
      - Stores elements in hash table
    - **TreeSet**
      - Stores elements in tree

---

```
// Fig. 22.12: SetTest.java
// Using a HashSet to remove duplicates.
import java.util.*;

public class SetTest {
    private static final String[] colors = { "red", "white", "blue", "green", "gray", "orange", "tan", "white", "cyan", "peach", "gray", "orange" };

    // create and output ArrayList
    public SetTest() {
        List list = new ArrayList( Arrays.asList( colors ) );
        System.out.println( "ArrayList: " + list );
        printNonDuplicates( list );
    }

    private void printNonDuplicates( Collection collection ) {
        // create a HashSet and obtain its iterator
        Set set = new HashSet( collection );
        Iterator iterator = set.iterator();

        System.out.println( "Nonduplicates are: " );
        Create HashSet from Collection object
    }
}
```

---

Outline

SetTest.java

Line 22
```java
27     while ( iterator.hasNext() )
28     System.out.print( iterator.next() + " " );
29    }
30    System.out.println();
31 }
32
33 public static void main( String args[] )
34 {
35    new SetTest();
36 }
37
38 } // end class SetTest
```

Arraylist: [red, white, blue, green, gray, orange, tan, white, cyan, peach, gray, orange]

Nonduplicates are:
red cyan white tan gray green orange blue peach

```java
1 // Fig. 22.13: SortedSetTest.java
2 // Using TreeSet and SortedSet.
3 import java.util.*;
4
5 public class SortedSetTest {
6   private static final String names[] = { "yellow", "green", "black", "tan", "grey", "white", "orange", "red", "green" };
7
8   // create a sorted set with TreeSet, then manipulate it
9   public SortedSetTest()
10   {
11     SortedSet tree = new TreeSet( Arrays.asList( names ) );
12     System.out.println( "set: " );
13     printSet( tree );
14
15     // get headSet based upon "orange"
16     System.out.print( "headSet ("orange"):  " );
17     printSet( tree.headSet( "orange" ) );
18
19     // get tailSet based upon "orange"
20     System.out.print( "tailSet ("orange"):  " );
21     printSet( tree.tailSet( "orange" ) );
22
23     // get first and last elements
24     System.out.println( "first: " + tree.first() );
25     System.out.println( "last : " + tree.last() );
26   }
27 }
```

Create TreeSet from names array

Use TreeSet method headSet to get TreeSet subset less than "orange"

Use TreeSet method tailSet to get TreeSet subset greater than "orange"

Methods first and last obtain smallest and largest TreeSet elements, respectively
22.8 Maps

- Map
  - Associates keys to values
  - Cannot contain duplicate keys
    - Called one-to-one mapping
  - Sets contain only values – whereas Maps contain keys and values
  - Several classes implement interface Map:
    - HashMap stores elements in hash tables
    - TreeMaps stores elements in trees
// Fig. 21.14: WordTypeCount.java
// Program counts the number of occurrences of each word in a string
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;

public class WordTypeCount extends JFrame {
    private JTextArea inputField;
    private JLabel prompt;
    private JTextArea display;
    private JButton goButton;
    private Map map;

    public WordTypeCount()
    {
        super( "Word Type Count" );
        inputField = new JTextArea( 3, 20 );
        map = new HashMap();
        goButton = new JButton( "Go" );
        goButton.addActionListener(
                      new ActionListener() { // inner class
            public void actionPerformed( ActionEvent event )
            {
                createMap();
                display.setText( createOutput() );
            }
        }); // end inner class
    }

    private void createMap()
    {
        String input = inputField.getText();
        String[] words = input.split(" ");
        for (String word : words)
        {
            if (map.containsKey(word))
            {
                map.put(word, map.get(word) + 1);
            }
            else
            {
                map.put(word, 1);
            }
        }
    }

    private String createOutput()
    {
        StringBuffer output = new StringBuffer();
        for (Map.Entry<String, Integer> entry : map.entrySet())
        {
            output.append(entry.getKey() + " : " + entry.getValue() + "
        }
        return output.toString();
    }

    public static void main(String[] args)
    {
        new WordTypeCount();
    }
}

```java
52    setSize( 400, 400 );
53    show();
54
55    } // end constructor
56
57    // create map from user input
58    private void createMap() {
59        String input = inputField.getText();
60        StringTokenizer tokenizer = new StringTokenizer( input );
61
62        while ( tokenizer.hasMoreTokens() ) {
63            String word = tokenizer.nextToken().toLowerCase(); // get word
64
65            // if the map contains the word
66            if ( map.containsKey( word ) ) {
67                Integer count = (Integer) map.get( word ); // get value
68
69                // increment value
70                map.put( word, new Integer( count.intValue() + 1 ) );
71            }
72            else { // otherwise add word with a value of 1 to map
73                map.put( word, new Integer( 1 ) );
74            }
75        } // end while
76
77    } // end method createMap

80    // create string containing map values
81    private String createOutput() {
82        StringBuffer output = new StringBuffer( "" );
83        Iterator keys = map.keySet().iterator();
84
85        // iterate through the keys
86        while ( keys.hasNext() ) {
87            Object currentKey = keys.next();
88
89            // output the key-value pairs
90            output.append( currentKey + "\t" +
91                map.get( currentKey ) + "\n" );
92        }
93
94        output.append( "size: " + map.size() + "\n" );
95        output.append( "isEmpty: " + map.isEmpty() + "\n" );
96
97        return output.toString();
98    } // end method createOutput
```

Use method `get` to retrieve a `Character` from `HashMap`.

Use method `put` to store a `Character` with an `Integer` key in `HashMap`.
Summary

- Java collections framework
  - prepackaged data structures and algorithms for manipulating these data structures
- A collection is an object that can hold references to objects.
- Collection interfaces declare operations for each type of collection.
- Important Constructs:
  - Class Arrays
  - Class Collections
  - Collection List
  - Collection Set
  - Collection Map
Part 8
Methods

• scope of declarations
• method overriding
• method overloading
• recursions

© 2003 Prentice Hall, Inc. All rights reserved.

6.1 Introduction

• Modules
  – Small pieces of a problem
    • e.g., divide and conquer
  – Facilitate design, implementation, operation and maintenance of large programs
6.2 Program Modules in Java

- Modules in Java
  - Methods
  - Classes
- Java API provides several modules
- Programmers can also create modules
  - e.g., programmer-defined methods
- Methods
  - Invoked by a method call
  - Returns a result to calling method (caller)
  - Similar to a boss (caller) asking a worker (called method) to complete a task

Fig. 6.1 Hierarchical boss-method/worker-method relationship.
Good Programming Practice

• Familiarize yourself with the rich collection of classes and methods provided by the Java API

java.sun.com/j2se/…/docs/api/index.html

and with the rich set of collections of classes available in various class libraries.

• Avoid reinventing the wheel
  – Reduce program development time and programming errors

• By rewriting existing Java API classes one rarely can improve the performance

Software Engineering Observation

• Promote software reusability: each method should perform a single, well-defined task; name of the method should express that task effectively.

• Small methods are easier to test and debug and promote software reusability.
6.3 Math-Class Methods

- Class `java.lang.Math`
  - Provides common mathematical calculations
  - Calculate the square root of 900.0:
    • `Math.sqrt( 900.0 )`
      - Method `sqrt` belongs to class `Math`
    • Dot (.) allows access to method `sqrt`
      - The argument 900.0 is located inside parentheses

- Class `Math` must not be explicitly imported.
  - Package `java.lang` is implicitly imported by the compiler.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>abs(x)</code></td>
<td>absolute value of x (this method also has <code>float</code>, <code>int</code> and <code>long</code> versions)</td>
<td><code>abs( 23.7 )</code> is 23.7</td>
</tr>
<tr>
<td><code>abs( 0.0 )</code></td>
<td><code>abs( 0.0 )</code> is 0.0</td>
<td></td>
</tr>
<tr>
<td><code>abs(-23.7)</code></td>
<td><code>abs(-23.7)</code> is 23.7</td>
<td></td>
</tr>
<tr>
<td><code>ceil(x)</code></td>
<td>rounds x to the smallest integer not less than x</td>
<td><code>ceil( 9.2 )</code> is 10.0</td>
</tr>
<tr>
<td><code>ceil(-9.8)</code></td>
<td><code>ceil(-9.8)</code> is -9.0</td>
<td></td>
</tr>
<tr>
<td><code>cos(x)</code></td>
<td>trigonometric cosine of x (x is in radians)</td>
<td><code>cos( 0.0 )</code> is 1.0</td>
</tr>
<tr>
<td><code>exp(x)</code></td>
<td>exponential method e^x</td>
<td><code>exp( 1.0 )</code> is 2.71828</td>
</tr>
<tr>
<td><code>exp( 2.0 )</code></td>
<td><code>exp( 2.0 )</code> is 7.38906</td>
<td></td>
</tr>
<tr>
<td><code>floor(x)</code></td>
<td>rounds x to the largest integer not greater than x</td>
<td><code>floor( 9.2 )</code> is 9.0</td>
</tr>
<tr>
<td><code>floor(-9.8)</code></td>
<td><code>floor(-9.8)</code> is -10.0</td>
<td></td>
</tr>
<tr>
<td><code>log(x)</code></td>
<td>natural logarithm of x (base e)</td>
<td><code>log(Math.E)</code> is 1.0</td>
</tr>
<tr>
<td><code>log(Math.E)</code></td>
<td><code>log(Math.E)</code> is 1.0</td>
<td></td>
</tr>
<tr>
<td><code>max(x, y)</code></td>
<td>larger value of x and y (this method also has <code>float</code>, <code>int</code> and <code>long</code> versions)</td>
<td><code>max( 2.3, 12.7 )</code> is 12.7</td>
</tr>
<tr>
<td><code>max(-2.3, -12.7)</code></td>
<td><code>max(-2.3, -12.7)</code> is -2.3</td>
<td></td>
</tr>
<tr>
<td><code>min(x, y)</code></td>
<td>smaller value of x and y (this method also has <code>float</code>, <code>int</code> and <code>long</code> versions)</td>
<td><code>min( 2.3, 12.7 )</code> is 2.3</td>
</tr>
<tr>
<td><code>min(-2.3, -12.7)</code></td>
<td><code>min(-2.3, -12.7)</code> is -12.7</td>
<td></td>
</tr>
<tr>
<td><code>pow(x, y)</code></td>
<td>x raised to the power y (xy)</td>
<td><code>pow(2.0, 7.0)</code> is 128.0</td>
</tr>
<tr>
<td><code>pow(9.0, 0.5)</code></td>
<td><code>pow(9.0, 0.5)</code> is 3.0</td>
<td></td>
</tr>
<tr>
<td><code>sin(x)</code></td>
<td>trigonometric sine of x (x is in radians)</td>
<td><code>sin(0.0)</code> is 0.0</td>
</tr>
<tr>
<td><code>sqrt(x)</code></td>
<td>square root of x</td>
<td><code>sqrt(900.0)</code> is 30.0</td>
</tr>
<tr>
<td><code>sqrt(9.0)</code></td>
<td><code>sqrt(9.0)</code> is 3.0</td>
<td></td>
</tr>
<tr>
<td><code>tan(x)</code></td>
<td>trigonometric tangent of x (x is in radians)</td>
<td><code>tan(0.0)</code> is 0.0</td>
</tr>
</tbody>
</table>

Fig. 6.2 Math-class methods.
6.4 Methods Declarations

• Methods
  – Allow programmers to modularize programs
    • Makes program development more manageable
    • Software reusability
    • Avoid repeating code
  – Local variables
    • Declared in method declaration
  – Parameters
    • Communicates information between methods via method calls
32     outputArea.setText( output ); // place results in JTextArea
33 }
34 // end method init
35
36 // square method declaration
37 public int square( int y )
38 {
39     return y * y; // return square of y
40 }
41 // end method square
42
43 // end class SquareIntegers

6.4 Method Declarations (cont.)

- General format of method declaration:

```
return-value-type  method-name( parameter1, parameter2, …, parameterN )
{
    declarations and statements
}
```

- Method can also return values:

```
return expression;
```

- Common Programming errors:
  - e.g. float x, y instead of float x, float y
  - data types of actual and formal parameters must match
  - Method header and calls must match in the number, type and order of parameters and arguments
Several Ways to Call a Method

- method name by itself, e.g. square(counter)
  Methods in a class declaration can call other methods in the same class declaration.
- reference to a method of an object
  g.drawLine(15,10,210,10)
- class name qualifying a method name
  (static method)
  Integer.parseInt(firstNumber)

Outline

1. Maximum.java
   Lines 13-18
   User inputs three Strings
   Convert Strings to doubles
   Method init passes doubles as arguments to method maximum

   MaximumTest.java
   1 // Fig. 6.4: MaximumTest.java
   2 // Finding the maximum of three floating-point numbers.
   3 import java.awt.Container;
   4
   5 import javax.swing.*;
   6
   7 public class MaximumTest extends JApplet {
   8  
   9  // initialize applet by obtaining user input and creating GUI
   10  public void init() {
   11  
   12  // obtain user input
   13  String s1 = JOptionPane.showInputDialog("Enter first floating-point value");
   14  String s2 = JOptionPane.showInputDialog("Enter second floating-point value");
   15  String s3 = JOptionPane.showInputDialog("Enter third floating-point value");
   16  
   17  // convert user input to double values
   18  double number1 = Double.parseDouble(s1);
   19  double number2 = Double.parseDouble(s2);
   20  double number3 = Double.parseDouble(s3);
   21  
   22  double max = maximum(number1, number2, number3); // method call
   23  
   24  // create JTextArea to display results
   25  JTextArea outputArea = new JTextArea();
   26  
   27  // display numbers and maximum value
   28  outputArea.setText("number1: " + number1 + 
   29    number2 + "\nnumber3: " + number3 + "\nmaximum is: " + max);
Method maximum returns value from method max of class Math

6.5 Argument Promotion

- Coercion of arguments
  - Forcing arguments to appropriate type to pass to method
    - e.g., `System.out.println( Math.sqrt( 4 ) );`
      - Evaluates `Math.sqrt( 4 )`
      - Then evaluates `System.out.println()`

- Promotion rules
  - Specify how to convert types without data loss
<table>
<thead>
<tr>
<th>Type</th>
<th>Valid promotions</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>None</td>
</tr>
<tr>
<td>float</td>
<td>double</td>
</tr>
<tr>
<td>long</td>
<td>float or double</td>
</tr>
<tr>
<td>int</td>
<td>long, float or double</td>
</tr>
<tr>
<td>char</td>
<td>int, long, float or double</td>
</tr>
<tr>
<td>short</td>
<td>int, long, float or double</td>
</tr>
<tr>
<td>byte</td>
<td>short, int, long, float or double</td>
</tr>
<tr>
<td>boolean</td>
<td>None (boolean values are not considered to be numbers in Java)</td>
</tr>
</tbody>
</table>

**Fig. 6.5** Allowed promotions for primitive types.

- Valid promotions from lower table entries to higher level table entries.
- Type conversion to lower table entries requires explicit cast operation.

```java
class result = square ((int) doubleValue);
```

### 6.6 Java API Packages

- **Packages**
  - Classes grouped into categories of related classes
  - Promotes software reuse
  - `import` statements specify classes used in Java programs
    - *e.g.*, `import javax.swing.JApplet;`
6.7 Random-Number Generation

- Java random-number generators
  - `Math.random()`
    - `1 + ( (int) ( Math.random() * 6 ) )`
      - Produces integers from 1 - 6
    - Use a seed for different random-number sequences
```java
public class RandomIntegers {
    public static void main(String[] args) {
        int value;
        String output = "";
        // loop 20 times
        for (int counter = 1; counter <= 20; counter++) {
            // pick random integer between 1 and 6
            value = 1 + (int) (Math.random() * 6);
            output += value + "  "; // append value to output
            // if counter divisible by 5, append newline
            if (counter % 5 == 0) {
                output += "\n";
            }
        } // end for
        JOptionPane.showMessageDialog(null, output, "20 Random Numbers from 1 to 6", JOptionPane.INFORMATION_MESSAGE);
        System.exit(0); // terminate application
    } // end main
} // end class RandomIntegers
```

Produce integers in range 1-6

Math.random returns doubles. We cast the double as an int
public static void main( String args[] )
{
    int frequency1 = 0, frequency2 = 0, frequency3 = 0,
            frequency4 = 0, frequency5 = 0, frequency6 = 0, face;
    // summarize results
    for ( int roll = 1; roll <= 6000; roll++ ) {
        // determine roll value and increment appropriate counter
        switch ( face ) {
            case 1:
                ++frequency1;
                break;
            case 2:
                ++frequency2;
                break;
            case 3:
                ++frequency3;
                break;
            case 4:
                ++frequency4;
                break;
            case 5:
                ++frequency5;
                break;
            case 6:
                ++frequency6;
                break;
        }
    }
    JTextArea outputArea = new JTextArea();
    outputArea.setText( "Face\tFrequency" +
            "\n1\t" + frequency1 +
            "\n2\t" + frequency2 +
            "\n3\t" + frequency3 +
            "\n4\t" + frequency4 +
            "\n5\t" + frequency5 +
            "\n6\t" + frequency6 );
    JOptionPane.showMessageDialog( null, outputArea,
            "Rolling a die 6000 Times", JOptionPane.INFORMATION_MESSAGE );
    System.exit( 0 );  // terminate application
} // end main
} // end class RollDie
6.9 Scope of Declarations (1)

• Scope
  – Portion of the program that can reference an entity by its name
  – Basic scope rules
    • Scope of a parameter declaration
    • Scope of a local-variable declaration
    • Scope of a label in a labeled **break** or **continue** statement
    • Scope of a local-variable declaration that appears in the initialization section of a **for** statement’s header
    • Scope of a method or field of a class
6.9 Scope of Declarations (2)

- Scope of a parameter declaration
  
  Body of the method in which the declaration appears. Scope of a constructor parameter declaration is the body of that constructor.

- Scope of a local variable declaration
  
  From the point at which the declaration appears in the block to the end of that block.

- Scope of a label in labeled break or continue statement
  
  The statement enclosed by the labeled statement (e.g. body of labeled statement).

6.9 Scope of Declarations (3)

- Scope of a local variable declaration in for statement
  
  body of the for statement

- Scope of a method or field of a class
  
  entire body of the class. This enables methods of a class to use simple names to call other methods declared in this class or inherited by that class and to access fields declared in the class.
Declarations and Scopes in Blocks

- Any block may contain variable or reference declarations.
- nested blocks: identifier in inner block with same name as identifier in outer block of the same method -> syntax error
- local variable or parameter with the same name as a field -> field is hidden until block terminates. shadowing

Avoid using the same names for fields and local variables.
Outline

Scoping.java

Line 42
Recreate variable x
and initialize it to 25

Lines 40-50
Method useLocal
uses local variable x

Re-create variable x
and initialize it to 25

Method useLocal
uses local variable x

Method useField
uses field x

Method useField
uses field x during each call

Outline

Scoping.java

Line 53
Recreate variable x
and initialize it to 25

Lines 53-61
Method useField
uses field x

Method useField
uses field x during each call

### 6.16 Methods of Class JApplet

- Java API defines several `JApplet` methods
  - Defining methods of Fig. 6.11 in a `JApplet` is called **overriding** those methods.
    - New method declaration overrides the inherited one.
  - Must use appropriate method header, otherwise default versions
  - New method declaration replaces default one

<table>
<thead>
<tr>
<th>Method</th>
<th>When the method is called and its purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>public void <code>init()</code></td>
<td>This method is called once by the applet container when an applet is loaded for execution. It performs initialization of an applet. Typical actions performed here are initializing fields, creating GUI components, loading sounds to play, loading images to display (see Chapter 19, Multimedia) and creating threads (see Chapter 16, Multithreading).</td>
</tr>
<tr>
<td>public void <code>start()</code></td>
<td>This method is called after the <code>init</code> method completes execution. In addition, if the browser user visits another Web site and later returns to the HTML page on which the applet resides, method <code>start</code> is called again. The method performs any tasks that must be completed when the applet is loaded for the first time and that must be performed every time the HTML page on which the applet resides is revisited. Typical actions performed here include starting an animation (see Chapter 19) and starting other threads of execution (see Chapter 16).</td>
</tr>
<tr>
<td>public void <code>paint(Graphics g)</code></td>
<td>This drawing method is called after the <code>init</code> method completes execution and the <code>start</code> method has started. It is also called every time the applet needs to be repainted. For example, if the user covers the applet with another open window on the screen and later uncovers the applet, the <code>paint</code> method is called. Typical actions performed here involve drawing with the <code>Graphics</code> object <code>g</code> that is passed to the <code>paint</code> method by the applet container.</td>
</tr>
<tr>
<td>public void <code>stop()</code></td>
<td>This method is called when the applet should stop executing—normally, when the user of the browser leaves the HTML page on which the applet resides. The method performs any tasks that are required to suspend the applet’s execution. Typical actions performed here are to stop execution of animations and threads.</td>
</tr>
<tr>
<td>public void <code>destroy()</code></td>
<td>This method is called when the applet is being removed from memory—normally, when the user of the browser exits the browsing session (i.e., closes all browser windows). The method performs any tasks that are required to destroy resources allocated to the applet.</td>
</tr>
</tbody>
</table>

*Fig. 6.11 JApplet methods that the applet container calls during an applet's execution.*
6.15 Method Overloading

- Method overloading
  - Several methods of the same name declared in the same class.
  - Different parameter set for each method
    - Number of parameters
    - Parameter types
  - Overloaded methods are distinguished by their signature
    - Method name and number and types of parameters
  - Overloaded methods are commonly used to perform similar tasks but on different types
  - Java compiler must decide which method to take.
  - Overloaded methods cannot be distinguished by return types if they have the same signature -> syntax error!

```
// Fig. 6.12: MethodOverload.java
// Using overloaded methods
import java.awt.Container;
import javax.swing.*;

public class MethodOverload extends JApplet {

  // create GUI and call each square method
  public void init() {
    JTextArea outputArea = new JTextArea();
    Container container = getContentPane();
    container.add(outputArea);
    outputArea.setText("The square of integer 7 is "+square(7)+
    "The square of double 7.5 is "+square(7.5));
  } // end method init

  // square method with int argument
  public int square(int intValue) {
    System.out.println("Called square with int argument: "+intValue);
    return intValue * intValue;
  } // end method square with int argument

  // square method with double argument
  public double square(double doubleValue) {
    System.out.println("Called square with double argument: "+doubleValue);
    return Math.pow(doubleValue, 2);
  } // end method square with double argument

} // end class MethodOverload
```
Overloaded method `square` receives a `double` as an argument.

```
31    // square method with double argument
32    public double square( double doubleValue )
33    {
34        System.out.println( "Called square with double argument: " +
35            doubleValue );
36
37        return doubleValue * doubleValue;
38
39    } // end method square with double argument
40
41    } // end class MethodOverload
```

```
127x187 MethodOverload.java:15: square(int) is already defined in MethodOverload
public double square( int y )
^ 1 error
```

Compiler cannot distinguish between methods with identical names and parameter sets.

```
1      // Fig. 6.13: MethodOverload.java
2      // Overloaded methods with identical signatures.
3      import javax.swing.JApplet;
4
5      public class MethodOverload extends JApplet {
6
7      // declaration of method square with int argument
8      public int square( int x )
9      {
10        return x * x;
11      }
12
13      // second declaration of method square
14      // with int argument causes syntax error
15      public double square( int y )
16      {
17        return y * y;
18      }
19
20    } // end class MethodOverload
```

```
MethodOverload.java:15: square(int) is already defined in MethodOverload
public double square( int y )
^ 1 error
```

Compiler cannot distinguish between methods with identical names and parameter sets.
6.12 Recursion

- Recursive method
  - Calls itself (directly or indirectly) through another method
  - Method knows how to solve only a base case
  - Method divides problem
    - Base case (simplest case)
    - Simpler problem
      - Method divides simpler problem until solvable
  - Recursive call = Recursive step
  - return statement to pass back result to caller of previous recursion.
  - sequence of smaller and smaller problems must converge to the base case
  - sequence of calls returns until original method call returns final result to caller.

\[ n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1 \]

(a) Sequence of recursive calls. (b) Values returned from each recursive call.

Final value = 120

5! = 5 \times 24 = 120 is returned
4! = 4 \times 6 = 24 is returned
3! = 3 \times 2 = 6 is returned
2! = 2 \times 1 = 2 is returned
1 is returned

import java.awt.*;
import javax.swing.*;

public class FactorialTest extends JApplet {
    JTextArea outputArea;

    // create GUI and calculate factorials of 0-10
    public void init() {
        outputArea = new JTextArea();
        Container container = getContentPane();
        container.add( outputArea );

        // calculate the factorials of 0 through 10
        for ( long counter = 0; counter <= 10; counter++ )
            outputArea.append( counter + "! = " + factorial( counter ) + "\n" );
    } // end method init

    // recursive declaration of method factorial
    public long factorial( long number ) {
        // base case
        if ( number <= 1 )
            return 1;
        // recursive step
        else
            return number * factorial( number - 1 );
    } // end method factorial

} // end class FactorialTest

// Fig. 6.15: FactorialTest.java
// Recursive factorial method.
import java.awt.*;
import javax.swing.*;

public class FactorialTest extends JApplet {
    JTextArea outputArea;

    // create GUI and calculate factorials of 0-10
    public void init() {
        outputArea = new JTextArea();
        Container container = getContentPane();
        container.add( outputArea );

        // calculate the factorials of 0 through 10
        for ( long counter = 0; counter <= 10; counter++ )
            outputArea.append( counter + "! = " + factorial( counter ) + "\n" );
    } // end method init

    // recursive declaration of method factorial
    public long factorial( long number ) {
        // base case
        if ( number <= 1 )
            return 1;
        // recursive step
        else
            return number * factorial( number - 1 );
    } // end method factorial

} // end class FactorialTest

// Fig. 6.15: FactorialTest.java
// Recursive factorial method.
import java.awt.*;
import javax.swing.*;

public class FactorialTest extends JApplet {
    JTextArea outputArea;

    // create GUI and calculate factorials of 0-10
    public void init() {
        outputArea = new JTextArea();
        Container container = getContentPane();
        container.add( outputArea );

        // calculate the factorials of 0 through 10
        for ( long counter = 0; counter <= 10; counter++ )
            outputArea.append( counter + "! = " + factorial( counter ) + "\n" );
    } // end method init

    // recursive declaration of method factorial
    public long factorial( long number ) {
        // base case
        if ( number <= 1 )
            return 1;
        // recursive step
        else
            return number * factorial( number - 1 );
    } // end method factorial

} // end class FactorialTest
Recursions: Common Programming Error

- omitting the base case
- incorrect based case or recursion step that never converges to base case

```java
public int nextRecursion( int number )
{
    if ( number == 3 )  return 1;
    else // recursive step
    return number * nextRecursion( number/2 );
}
```

- input parameter out of range (e.g. negative instead of positive value)
  - cover with correct typing

Commonly recursion errors yield infinite recursion eventually exhausting memory.

6.13 Example Using Recursion: The Fibonacci Series

- Fibonacci series
  - Each number in the series is sum of two previous numbers
    - e.g., 0, 1, 1, 2, 3, 5, 8, 13, 21…

    ```java
    fibonacci(0) = 0
    fibonacci(1) = 1
    fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2)
    ```

    - fibonacci(0) and fibonacci(1) are base cases
      - Golden ratio (golden mean)
// Fig. 6.16: FibonacciTest.java
// Recursive fibonacci method.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class FibonacciTest extends JApplet implements ActionListener {
    JLabel numberLabel, resultLabel;
    JTextField numberField, resultField;

    // set up applet's GUI
    public void init() {
        // obtain content pane and set its layout to FlowLayout
        Container container = getContentPane();
        container.setLayout( new FlowLayout() );

        // create numberLabel and attach it to content pane
        numberLabel = new JLabel( "Enter an integer and press Enter" );
        container.add( numberLabel );

        // create numberField and attach it to content pane
        numberField = new JTextField( 10 );
        container.add( numberField );

        // register this applet as numberField's ActionListener
        numberField.addActionListener( this );

        // create resultLabel and attach it to content pane
        resultLabel = new JLabel( "Fibonacci value is" );
        container.add( resultLabel );

        // create numberField, make it uneditable
        // and attach it to content pane
        resultField = new JTextField( 15 );
        resultField.setEditable( false );
        container.add( resultField );
    } // end method init

    // obtain user input and call method fibonacci
    public void actionPerformed( ActionEvent event ) {
        long number, fibonacciValue;

        // obtain user's input and convert to long
        number = Long.parseLong( numberField.getText() );

        showStatus( "Calculating ..." );

        // calculate fibonacci value for number user input
        fibonacciValue = fibonacci( number );

        // indicate processing complete and display result
        showStatus( "Done." );
        resultField.setText( Long.toString( fibonacciValue ) );
    } // end method actionPerformed

    // recursive fibonacci method
    public long fibonacci( long number ) {
        // base case: return 0 or 1
        if ( number == 0 ) return 0;
        else if ( number == 1 ) return 1;

        // recursive case: return fibonacci of previous two numbers
        return fibonacci( number - 1 ) + fibonacci( number - 2 );
    } // end method fibonacci
}

// method to display a status message in the console
public void showStatus( String status ) {
    System.out.println( status );
}
public long fibonacci(long n) {
    // base case
    if (n == 0 || n == 1)
        return n;
    // recursive step
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
} // end method fibonacci

} // end class FibonacciTest
Fig. 6.17 Set of recursive calls for \texttt{fibonacci(3)}. 
6.14 Recursion vs. Iteration

• Iteration
  – Uses repetition structures (for, while or do...while)
  – Repetition through explicitly use of repetition structure
  – Terminates when loop-continuation condition fails
  – Controls repetition by using a counter

• Recursion
  – Uses selection structures (if, if...else or switch)
  – Repetition through repeated method calls
  – Terminates when base case is satisfied
  – Controls repetition by dividing problem into simpler one

6.14 Recursion vs. Iteration (cont.)

• Recursion
  – More overhead than iteration
  – More memory intensive than iteration
  – Can also be solved iteratively
  – Often can be implemented with only a few lines of code
Summary

• modules in Java: classes and methods
• software reuse: methods for modularizing programs
• scope of declarations
• method overriding
  New method declaration overrides the inherited one.
• method overloading
  Several methods of the same name declared in the same class.

• recursions versus iteration
Part 10
Object Oriented Programming

Traditionelle Softwareentwicklung

- **Wasserfallmodell**
  - Dominierte lange die Softwareentwicklung
  - Weite Verbreitung in vielen Varianten

- **Hauptprobleme:**
  - Starre Unterteilung von Softwareprojekten
  - Sprunghafte Phasenübergänge
  - Spätes Erkennen von Designfehlern (*late design breakage*)
  - Gefahr, Anforderungen nicht zu erfüllen
**Traditionelle Softwareentwicklung**

- **Strukturierte Analyse und Design (SA/SD)**
  - Basis sind Analysetechniken, die zwischen 1965 u. 1975 entwickelt wurden.
  - Im Mittelpunkt steht die **Modellierung von Prozessen** durch *Funktionen*.
  - sehr weit verbreitet (klassisches SE)
  - Schwerpunkt: Implementierung und Test
  - schwierige Wartung

**Kosten der Fehlerkorrektur**

- **Definition**
  - 1 x

- **Implementation**
  - 1,5-6 x

- **After release**
  - 60-100 x
Objektorientierte Softwareentwicklung

Ende der 80er Jahre erfolgt ein Paradigmenwechsel im Software Engineering: **Objektorientierung.**

- **Objektorientierte Analyse und Design (OOA/OOD)**
  - Basierend auf OO-Technologie (ab ca. 1985)
  - Zentral sind **Objektmodelle** (Vereinigung von Daten- und Prozessmodellen)
  - Schwerpunkt: Analyse und Design
  - **leichtere** Wartung

![Diagramm der Aufwände von OOA/OOD und SA/SD](image)

- **OOA/OOD versus SA/SD**
  - **OOA/OOD**
    - Zerlegung anhand von Konzepten (Objekten)
  - **SA/SD**
    - Zerlegung anhand von Prozessen (Funktionen)
Introduction: Procedural Programming

• Procedural programming language
  – C, Pascal, Fortran, ...
  – operations-oriented
  – operations to perform a task grouped into functions
  – functions form a program
  – data to support functions
  – data and operations (procedures, functions) are separated
  – How to realize a functionality?
  – Functions are units of programming

Introduction: Object-Oriented Programming

• Object-oriented programming language
  – Java, Smalltalk, C++
  – Object-oriented
  – Data and operations (methods) define a unit (class)
  – Classes to encapsulate data (attributes) and methods (behavior)
  – Encapsulation for hiding implementation
  – What (functionality) should be realized?
  – Classes are units of programming
    • members
      – fields (variables)
      – methods for manipulating fields
    • functions, or methods, are encapsulated in classes
Key Object Oriented Concepts (1)

• Classes and Objects
  – A class describes the characteristics (variables) and behavior (methods) of its objects.
  – A class declaration defines the **variables and methods** of an object.
  – A class declaration defines a new reference type.

```java
class Person { // class declaration
    String name; // variable (field)
    private String svnr; // ...
    ...
    public Person(String name, String svnr) { // constructor
        this.name = name; this.svnr = svnr;
    }
    public int getAlter() (...) // method
}```

Key Object Oriented Concepts (2)

• State of an object
  – defined by the contents of its variables

• Behavior of an object
  – defined through the object’s methods.

• Object interaction
  – Objects interact with each other through exchanging messages (method invocation).
• Classes and Objects
  – Objects are instances of a class.
  – Objects are created through the invocation of a constructor (new Operator).

```java
// create instance (object) of class Person
Person hugo = new Person("Hugo", "1121030333");
```

– Access to variables and methods of an object is achieved through the "." Operator

```java
String name = hugo.name; // access variable
int a = hugo.getAge(); // call method
```

### Classes and Objects: UML Diagrams
Key Object Oriented Concepts (4)

- **Encapsulation (Information hiding)**

  - The internal structure of an object should not be accessible or visible to the outside world.
  - Program that is using objects of a specific class should be independent of the implementation of this class.
  - Know what an object can do, not how it does it.
  - Java access modifiers: public, private, protected
  - Information hiding increases the level of independence.
  - Independence of modules is important for large systems and maintenance.

```java
class Person {
    ...
    private String svnr; // not accessible outside
    public int getSvnr() {...} // access method
}
```

---

Java Language Constructs - Classes

- Class declaration
- Variables
- Constructor
- Methods
Implementing a Time Abstract Data Type with a Class

• Java simplifies creation of abstract data types
  – hide implementation details
  – internal implementation can be changed without changing interfaces (method signature)

• We introduce classes `Time1` and `TimeTest`
  – `Time1.java` declares class `Time1`
  – `TimeTest.java` declares class `TimeTest`
  – `public` classes must be declared in separate files
  – Class `Time1` will not execute by itself
    • Does not have method `main`
    • `TimeTest`, which has method `main`, creates (instantiates) and uses `Time1` object
8.2 Implementing a Time Abstract Data Type with a Class (cont.)

• Every Java class must extend another class
  – Time1 extends java.lang.Object
  – If class does not explicitly extend another class
    • class implicitly extends Object

• Class constructor
  – Same name as class
  – Initializes instance variables of a class object
  – Called when program instantiates an object of that class
  – Can take arguments, but cannot return values
  – Class can have several constructors, through overloading
  – Class Time1 constructor(lines 12-15)
TimeTest1.java

public class TimeTest1 {
    public static void main(String args[]) {
        Time1 time = new Time1(); // calls Time1 constructor

        String output = "The initial universal time is: " + time.toUniversalString() + "The initial standard time is: " + time.toStandardString();

        time.setTime(13, 27, 6);
        output += "Universal time after setTime is: " + time.toUniversalString() + "Standard time after setTime is: " + time.toStandardString();

        time.setTime(99, 99, 99);
        output += "After attempting invalid settings: " + time.toUniversalString() + "Standard time: " + time.toStandardString();

        JOptionPane.showMessageDialog(null, output, "Testing Class Time1", JOptionPane.INFORMATION_MESSAGE);

        System.exit(0);
    }
}

TimeTest1 interacts with Time1 by calling Time1 public methods

// Fig. 8.2: TimeTest1.java
// Class TimeTest1 to exercise class Time1.
import javax.swing.JOptionPane;

test1.java
Line 9
Declare and create instance of class
Time1 by calling Time1 constructor
Lines 12-26
TimeTest1 interacts with Time1
by calling Time1 public methods

Testing Class Time1

The initial universal time is: 01:00:00
The initial standard time is: 12:00:00 AM

Universal time after setTime is: 13:27:06
Standard time after setTime is: 12:00:00 PM

After attempting invalid settings:
Universal time: 00:00:00
Standard time: 12:00:00 AM

OK
Main concepts to be covered

- fields
- constructors
- methods
- parameters
- object interaction
- garbage collection
- packages

Basic class structure

```java
public class TicketMachine {
    // Inner part of the class omitted.
}

public class ClassName {
    // Fields
    // Constructors
    // Methods
}
```

The outer wrapper of TicketMachine

The contents of a class
Fields

- Fields store values for an object.
- They are also known as instance variables.
- Fields define the state of an object.

```java
public class TicketMachine {
    private int price;
    private int balance;
    private int total;

    // Constructor and methods omitted.
}
```

Class Scope

- Class scope
  - Class variables and methods
  - Members are accessible to all class methods
  - Members can be referenced by name
    - objectReferenceName.objectMemberName
  - Shadowed (hidden) class variables
    - this.variableName
8.4 Controlling Access to Members

- Member access modifiers
  - Control access to class’s variables and methods
    - public
      - Variables and methods accessible to clients of the class
    - private
      - Variables and methods not accessible to clients of the class
- Precede every field and method declaration with access modifier. Rule of thumb:
  - fields as private
  - methods as public
- private methods: utility methods
  - can be called only by other methods of the same class
- public fields is uncommon and dangerous programming!
Referring to the Current Object’s Members with this

- Keyword **`this`** (*this reference*)
  - Allows an object to refer to itself
- **Syntax error**: non-constructor method tries to invoke constructor via this reference.
- Constructor can call other methods
  - Warning: instance variables may not be consistent yet

---

```java
// Fig. 8.4: ThisTest.java
// Using the this reference to refer to instance variables and methods.
import java.text.DecimalFormat;

public class ThisTest {
    public static void main( String args[] )
    {
        SimpleTime time = new SimpleTime( 12, 30, 19 );
        JOptionPane.showMessageDialog( null, time.buildString(),
            "Demonstrating the \"this\" Reference",
            JOptionPane.INFORMATION_MESSAGE );
        System.exit( 0 );
    }
}

class SimpleTime {
    private int hour;
    private int minute;
    private int second;
}
```

---

ThisTest.java

```java
// Fig. 8.4: ThisTest.java
// Using the this reference to refer to instance variables and methods.
import java.text.DecimalFormat;

public class ThisTest {
    public static void main( String args[] )
    {
        SimpleTime time = new SimpleTime( 12, 30, 19 );
        JOptionPane.showMessageDialog( null, time.buildString(),
            "Demonstrating the \"this\" Reference",
            JOptionPane.INFORMATION_MESSAGE );
        System.exit( 0 );
    }
}

class SimpleTime {
    private int hour;
    private int minute;
    private int second;
}
```
Initializing Class Objects: Constructors

- Class constructor
  - Same name as class
  - Initializes instance variables of a class object
  - Call class constructor to instantiate object of that class

  ```java
  new ClassName( argument1, argument2, ..., argumentN );
  ```

  - `new` indicates that new object is created
  - `ClassName` indicates type of object created
  - `arguments` specifies constructor argument values
  - Constructors never return values and cannot specify a return type
  - Other methods with same name as class can return values but they are not constructors and will not be called when object is created.
  - Constructors are normally declared public
Using Overloaded Constructors

- Overloaded constructors
  - Methods (in same class) may have same name
  - Must have different parameter lists

```java
// Fig. 8.5: Time2.java
// Time2 class declaration with overloaded constructors.
import java.text.DecimalFormat;

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

    // Time2 constructor initializes each instance variable to zero;
    // ensures that Time object starts in a consistent state
    public Time2() {
        this( 0, 0, 0 ); // invoke Time2 constructor with three arguments
    }

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

    // 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
    }
}
```
// Time2 constructor: hour, minute and second supplied
public Time2( int h, int m, int s )
{
    setTime( h, m, s ); // invoke setTime to validate time
}

// Time2 constructor: another Time2 object supplied
public Time2( Time2 time )
{
    // invoke Time2 constructor with three arguments
    this( time.hour, time.minute, time.second );
}

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

// convert to String in universal-time format
public String toUniversalString()
{
    DecimalFormat twoDigits = new DecimalFormat( "00" );
    return twoDigits.format( hour ) + ":" +
    twoDigits.format( minute ) + ":" + twoDigits.format( second );
}

// convert to String in standard-time format
public String toStandardString()
{
    DecimalFormat twoDigits = new DecimalFormat( "00" );
    return ( (hour == 12 || hour == 0) ? 12 : hour % 12 ) + ":" +
    twoDigits.format( minute ) + ":" + twoDigits.format( second ) +
    ( hour < 12 ? " AM" : " PM" );
}

// end class Time2
// Fig. 8.6: TimeTest3.java
// Overloaded constructors used to initialize Time2 objects.
import javax.swing.*;

public class TimeTest3 {
    public static void main(String args[]) {
        // Instantiate each Time2 reference using a different constructor

        Time2 t1 = new Time2();             // 00:00:00
        Time2 t2 = new Time2(2);            // 02:00:00
        Time2 t3 = new Time2(21, 34);       // 21:34:00
        Time2 t4 = new Time2(12, 25, 42);   // 12:25:42
        Time2 t5 = new Time2(27, 76, 99);   // 00:00:00
        Time2 t6 = new Time2(t4);           // 12:25:42

        String output = "Constructed with: " +
                         "\nt1: all arguments defaulted" +
                         "\n" + t1.toUniversalString() +
                         "\n" + t1.toStandardString();

        output += "\nt2: hour specified; minute and second defaulted" +
                         "\n" + t2.toUniversalString() +
                         "\n" + t2.toStandardString();

        output += "\nt3: hour and minute specified; second defaulted" +
                         "\n" + t3.toUniversalString() +
                         "\n" + t3.toStandardString();

        output += "\nt4: hour, minute and second specified" +
                         "\n" + t4.toUniversalString() +
                         "\n" + t4.toStandardString();

        output += "\nt5: all invalid values specified" +
                         "\n" + t5.toUniversalString() +
                         "\n" + t5.toStandardString();

        output += "\nt6: Time2 object t4 specified" +
                         "\n" + t6.toUniversalString() +
                         "\n" + t6.toStandardString();

        JOptionPane.showMessageDialog(null, output,
                                         "Overloaded Constructors", JOptionPane.INFORMATION_MESSAGE);

        System.exit(0);
    } // end main
}
} // end class TimeTest3
Using Set and Get Methods

- Accessor method (“get” method)
  - public method
  - Allow clients to read private data
- Mutator method (“set” method)
  - public method
  - Allow clients to modify private data

Accessor methods

- Methods implement the behavior of objects.
- Accessors provide information about an object.
- Methods have a structure consisting of a header and a body.
  - The header defines the method’s signature.
    public int getPrice()
  - The body encloses the method’s statements.
Accessor methods

Mutator methods

- Have a similar method structure: header and body.
- Used to mutate (i.e., change) an object’s state.
- Achieved through changing the value of one or more fields.
  - Typically contain assignment statements.
  - Typically receive parameters.
**Mutator methods**

```
public void insertMoney(int amount)
{
    balance += amount;
}
```

<table>
<thead>
<tr>
<th>access modifier</th>
<th>return type (void)</th>
<th>method name</th>
<th>parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>void</td>
<td>insertMoney</td>
<td>int amount</td>
</tr>
</tbody>
</table>

- **Fields** are one sort of variable.
  - They store values through the life of an object.
  - They are accessible throughout the class.
- **Methods** can include shorter-lived variables.
  - They exist only as long as the method is being executed.
  - They are only accessible from within the method.
  - Local variables cannot have modifier

**Local variables**

- Fields are one sort of variable.
  - They store values through the life of an object.
  - They are accessible throughout the class.
- Methods can include shorter-lived variables.
  - They exist only as long as the method is being executed.
  - They are only accessible from within the method.
  - Local variables cannot have modifier
Local variables

public int refundBalance()
{
    int amountToRefund;
    amountToRefund = balance;
    balance = 0;
    return amountToRefund;
}

Avoid method-parameter names or local variable names that conflict with field names.

// Fig. 8.7: Time3.java
// Time3 class declaration with set and get methods.
import java.text.DecimalFormat;

public class Time3 {
    private int hour;     // 0 - 23
    private int minute;   // 0 - 59
    private int second;   // 0 - 59

    // Time3 constructor initializes each instance variable to zero;
    // ensures that Time object starts in a consistent state
    public Time3()
    { this( 0, 0, 0 ); // invoke Time3 constructor with three arguments }

    // Time3 constructor: hour supplied, minute and second defaulted to 0
    public Time3( int h )
    { this( h, 0, 0 ); // invoke Time3 constructor with three arguments }

    // Time3 constructor: hour and minute supplied, second defaulted to 0
    public Time3( int h, int m )
    { this( h, m, 0 ); // invoke Time3 constructor with three arguments }
Set methods allow objects to manipulate private variables

Get methods allow objects to read private variables
// get second value
public int getSecond()
{
    return second;
}

// convert to String in universal-time format
public String toUniversalString()
{
    DecimalFormat twoDigits = new DecimalFormat( "00" );

    return twoDigits.format( getHour() ) + ":" +
    twoDigits.format( getMinute() ) + ":" +
    twoDigits.format( getSecond() );
}

// convert to String in standard-time format
public String toStandardString()
{
    DecimalFormat twoDigits = new DecimalFormat( "00" );

    return ( ( getHour() == 12 || getHour() == 0 ) ?
    12 : getHour() % 12 ) + ":" + twoDigits.format( getMinute() ) +
    ":" + twoDigits.format( getSecond() ) +
    ( getHour() < 12 ? " AM" : " PM" );
}

} // end class Time3
A digital clock

11:03

Abstraction and modularization

- **Abstraction** is the ability to ignore details of parts to focus attention on a higher level of a problem.
- **Modularization** is the process of dividing a whole into well-defined parts, which can be built and examined separately, and which interact in well-defined ways.
Modularizing the clock display

One four-digit display?

Or two two-digit displays?

Implementation - NumberDisplay

```java
public class NumberDisplay {
    private int limit;
    private int value;

    Constructor and methods omitted.
}
```
public class ClockDisplay
{
    private NumberDisplay hours;
    private NumberDisplay minutes;

    Constructor and
    methods omitted.
}
Class diagram

```
 SomeObject obj;
```

Object type

```
 int i;
```

Primitive types vs. object types

Object type

```
32
```

Primitive type
**Primitive types vs. object types**

SomeObject\ a; SomeObject\ b;

\[ b = a; \]

int\ a; int\ b;

32 32

**Source code: NumberDisplay**

```java
class NumberDisplay {
    public NumberDisplay(int rollOverLimit) {
        limit = rollOverLimit;
        value = 0;
    }

    public void increment() {
        value = (value + 1) % limit;
    }
}
```
Source code: NumberDisplay

```java
public String getDisplayValue()
{
    if(value < 10)
        return "0" + value;
    else
        return "" + value;
}
```

Objects creating objects

```java
public class ClockDisplay
{
    private NumberDisplay hours;
    private NumberDisplay minutes;
    private String displayString;

    public ClockDisplay()
    {
        hours = new NumberDisplay(24);
        minutes = new NumberDisplay(60);
        updateDisplay();
    }
}
```
public void timeTick()
{
    minutes.increment();
    if(minutes.getValue() == 0) {
        // it just rolled over!
        hours.increment();
    }
    updateDisplay();
}

/**
 * Update the internal string that
 * represents the display.
 */
private void updateDisplay()
{
    displayString =
        hours.getDisplayValue() + "::" +
        minutes.getDisplayValue();
}
Objects creating objects

in class NumberDisplay:
    public NumberDisplay(int rollOverLimit);

formal parameter

in class ClockDisplay:
    hours = new NumberDisplay(24);

actual parameter
Method calls

• internal method calls

   updateDisplay();
   …

   private void updateDisplay()

• external method calls

   minutes.increment();

Class `Date` encapsulates data that describes date

Date constructor instantiates Date object based on specified arguments
else { // month is invalid
    System.out.println( "Invalid month (" + testMonth + 
    ");
    return 1; // maintain object in consistent state
}

else { // month is invalid
    System.out.println( "Invalid month (" + testMonth + 
    ");
    return 1; // maintain object in consistent state
}

// utility method to confirm proper day value based on month and year
private int checkDay( int testDay )
{
    int daysPerMonth[] =
    { 0, 31, 28, 31, 30, 31, 30, 31, 30, 31, 30, 31, 31 };

    // check if day in range for month
    if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
        return testDay;

    // check for leap year
    if ( month == 2 && testDay == 29 && ( year % 400 == 0 ||
        year % 4 == 0 && year % 100 != 0 ) )
        return testDay;

    System.out.println( "Invalid day (" + testDay + ");
    return 1; // maintain object in consistent state
}

// return a String of the form month/day/year
public String toDateString()
{
    return month + "/" + day + "/" + year;
}

} // end class Date
Employee.java

Lines 7-8
Employee is composed of two references to Date objects

EmployeeTest.java

Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
8.10 Garbage Collection

- Garbage collection
  - Java runtime system returns memory of objects which are no longer needed (objects without references to it).
  - Java performs this automatically
    - object marked for garbage collection if no references to object
  - no guarantee that the garbage collector will be called
  - garbage collector runs with lower priority in parallel with a given program
- Finalizer method
  - Returns resources (e.g. files or network connections) to system
  - executed before garbage collector reclaims object
  - Java provides method `finalize`
    - Defined in `java.lang.Object`
    - Receives no parameters
    - cannot be overloaded
    - Returns `void`

8.11 Static Class Members

- `static` keyword
  - `static` class variable
    - Class-wide information
      - All class objects share same data
- Access to a class’s `public static` members
  - Qualify the member name with the class name and a dot (.)
    - e.g., Math.random()
- static class methods and variables exist and can be used, even if no objects have been instantiated.
- static methods cannot access non-static class members.
- static methods have no this reference.
// Fig. 8.12: Employee.java
// Employee class declaration.
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.println( "Employee constructor: " +
            firstName + " " + lastName );
    }

    // subtract 1 from static count when garbage collector
    // calls finalize to clean up object and output String
    // indicating that finalize was called
    protected void finalize()
    {
        --count; // decrement static count of employees
        System.out.println( "Employee finalizer: " +
            firstName + " " + lastName + "; count = " + count );
    }

    // get first name
    public String getFirstName()
    { return firstName; }

    // get last name
    public String getLastName()
    { return lastName; }

    // static method to get static count value
    public static int getCount()
    { return count; }
}
} // end class Employee

Employee objects share one instance of count

Lines 23-28
Called when Employee is marked for garbage collection

static method accesses static variable count

Lines 43-46
EmployeeTest can invoke Employee static method, even though Employee has not been instantiated.

EmployeeTest.java

// Fig. 8.13: EmployeeTest.java
// Test Employee class with static class variable, static class method, and dynamic memory.
import javax.swing.*;

public class EmployeeTest {

    public static void main( String args[] ) {
        // prove that count is 0 before creating Employees
        String output = "Employees before instantiation: " + Employee.getCount();

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

        // prove that count is 2 after creating two Employees
        output += "Employees after instantiation: " + e1.getCount() + "" + e2.getCount() + "" + Employee.getCount();

        // get names of Employees
        output += "Employee 1: " + e1.getFirstName() + " " + e1.getLastName() + "Employee 2: " + e2.getFirstName() + " " + e2.getLastName();

        // decrement reference count for each Employee object; in this example, there is only one reference to each Employee, so these statements mark each Employee object for garbage collection
        e1 = null;
        e2 = null;

        System.gc(); // suggest call to garbage collector

        // show Employee count after calling garbage collector; count displayed may be 0, 1 or 2 based on whether garbage collector executes immediately and number of Employee objects collected
        output += "Employees after System.gc(): " + Employee.getCount();

        JOptionPane.showMessageDialog( null, output, "Static Members", JOptionPane.INFORMATION_MESSAGE );
        System.exit( 0 );
    }
} // end class EmployeeTest

Calls Java’s automatic garbage-collection mechanism

Employee constructor: Susan Baker
Employee constructor: Bob Jones
Employee finalizer: Susan Baker; count = 1
Employee finalizer: Bob Jones; count = 0

Calls Java’s automatic garbage-collection mechanism

Employee constructor: Susan Baker
Employee constructor: Bob Jones
Employee finalizer: Susan Baker; count = 1
Employee finalizer: Bob Jones; count = 0
### 8.12 Final Instance Variables

- **final keyword**
  - Indicates that variable is not modifiable
    - Any attempt to modify `final` variable results in error
      
      ```java
      private final int INCREMENT = 5;
      ```
  - Declares variable `INCREMENT` as a constant and initializes the variable
    - declaration and initialization can be separated

      ```java
      private final int INCREMENT;
      ...
      INCREMENT = 5; // must be in a constructor
      ```
  - syntax error if
    - final instance variables without initialization
    - final instance variables is tried to be modified after initialization
final keyword declares `INCREMENT` as constant

final variable `INCREMENT` must be initialized before using it

If statement 41 is outcommented

IncrementTest.java:40: variable INCREMENT might not have been initialized
```java
Increment.java
Line 36
final keyword declares INCREMENT as constant
Line 41
final variable INCREMENT must be initialized before using it
```
### 8.13 Creating Packages

- We can import packages into programs
  - Group of related classes and interfaces
  - Help manage complexity of application components
  - Facilitate software reuse
  - Provide convention for unique class names
    - Popular package-naming convention
      - Reverse Internet domain name
        - e.g., com.deitel

- Steps for creating a reusable class
  - declare public class
  - choose a package name
  - add package name to the source code of class
  - compile the class (placed in appropriate directory structure)
  - import reusable class into a program for reuse

```java
// Fig. 8.16: Time1.java
// Time1 class declaration maintains the time
package com.deitel.jhtp5.ch08;
import java.text.DecimalFormat;

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

    // Time1 constructor initializes each instance variable to zero;
    // ensures that each Time1 object starts in a consistent state
    public Time1() {
        setTime(0, 0, 0);
    }

    // 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);
        minute = ((m >= 0 && m < 60) ? m : 0);
        second = ((s >= 0 && s < 60) ? s : 0);
    }

    // Fig. 8.16: Time1.java
    // Time1 class declaration maintains the time
    // import class DecimalFormat from package java.text

    // Class Time1 is placed in this package
    // Class Time1 is in directory com/deitel/jhtp5/ch08

    class Time1 will be placed in directory ch08 in com/deitel/jhtp5
    compiling with “javac –d” generates appropriate directory structure if it does not exist.
```
public String toUniversalString()
{
    DecimalFormat twoDigits = new DecimalFormat("00");
    return twoDigits.format(hour) + ";" +
    twoDigits.format(minute) + ";" + twoDigits.format(second);
}

public String toStandardString()
{
    DecimalFormat twoDigits = new DecimalFormat("00");
    return (hour == 12 || hour == 0) ? 12 : hour % 12) + ";" +
    twoDigits.format(minute) + ";" + twoDigits.format(second) +
    (hour < 12 ? " AM" : " PM");
}

} // end class Time1
```java
27    // set time with invalid values; append updated time to output
28    time.setTime( 99, 99, 99 );
29    output += "\n\nAfter attempting invalid settings: " +
30    "\nUniversal time: " + time.toUniversalString() +
31    "\nStandard time: " + time.toStandardString();
32
33    JOptionPane.showMessageDialog( null, output,
34    "Testing Class Time1", JOptionPane.INFORMATION_MESSAGE );
35
36    System.exit( 0 );
37
38    } // end main
39
40    } // end class TimeTest1
```

8.14 Package Access

- **Package access**
  - Variable or method without access modifier (public, protected or private)
  - default access modifier: package access

- **Program with one class declaration**
  - no effect

- **Program that uses multiple classes from the same package**
  - classes can access each other’s package access methods and fields through proper object references
import javax.swing.JOptionPane;

public class PackageDataTest {
    public static void main( String[] args ) {
        PackageData packageData = new PackageData();

        // append String representation of packageData to output
        String output = "After instantiation:
" + packageData.toPackageDataString();

        // change package access data in packageData object
        packageData.number = 77;
        packageData.string = "Goodbye";

        // append String representation of packageData to output
        output += "After changing values:
" + packageData.toPackageDataString();

        JOptionPane.showMessageDialog( null, output, "Package Access", JOptionPane.INFORMATION_MESSAGE );

        System.exit( 0 );
    }
}

// class with package access instance variables
class PackageData {
    int number;     // package-access instance variable
    String string;  // package-access instance variable

    // constructor
    public PackageData() {
        number = 0;
        string = "Hello";
    }

    // return PackageData object String representation
    public String toPackageDataString() {
        return "number: " + number + "    string: " + string;
    }
}

No access modifier, so class has package-access variables.
• Class bodies contain fields, constructors and methods.
• Fields store values that determine an object’s state.
• Constructors initialize objects.
• Methods implement the behavior of objects.

• Fields, parameters and local variables are all variables.
• Fields persist for the lifetime of an object.
• Parameters are used to receive values into a constructor or method.
• Local variables are used for short-lived temporary storage.
• Classes can be grouped to packages for software reuse.
Part 11
Object Oriented Programming: Inheritance

Main concepts to be covered

• Superclasses and Subclasses
• protected Members
• Relationship between Superclasses and Subclasses
• Case Study: Three-Level Inheritance Hierarchy
• Constructors and Finalizers in Subclasses
• Software Engineering with Inheritance
Introduction: Inheritance

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

Central object oriented concepts (1)

- Inheritance
  - Derive subclasses from existing classes (Class hierarchy).
  - The subclass inherits the variables and methods of a superclass.
  - A subclass may define new variables and methods.
  - Methods of a superclass can be overwritten in a subclass.

```java
class Person { // superclass
    String name;
    private String svnr;
    public int getAlter {...}
}
class Student extends Person { // subclass
    String matrNr;
}
```
Central object oriented concepts (2)

- **Class Object**
  - Class Object implicitly is a superclass of any other class.
  - Class Object provides methods which can be used by any other class.

- **Example:**
  - The Java Collection API (java.util) uses class Object to support data structures (Vector, Stack, LinkedList, Hashtable, ...) for management of objects derived from arbitrary classes.

Class Hierarchy

- Direct superclass
  - Inherited explicitly (one level up hierarchy)
- Indirect superclass
  - Inherited two or more levels up hierarchy
- Single inheritance
  - Inherits from one superclass
- Multiple inheritance
  - Inherits from multiple superclasses
  - Java does not support multiple inheritance
The DoME example

"Database of Multimedia Entertainment"

- stores details about CDs and videos
  - CD: title, artist, # tracks, playing time, got-it, comment
  - Video: title, director, playing time, got-it, comment
- allows (later) to search for information or print lists
DoME classes

<table>
<thead>
<tr>
<th>CD</th>
<th>Video</th>
</tr>
</thead>
<tbody>
<tr>
<td>title</td>
<td>title</td>
</tr>
<tr>
<td>artist</td>
<td>director</td>
</tr>
<tr>
<td>numberOfTracks</td>
<td>playingTime</td>
</tr>
<tr>
<td>playingTime</td>
<td>goItt</td>
</tr>
<tr>
<td>goItt</td>
<td>comment</td>
</tr>
<tr>
<td>comment</td>
<td>setComment</td>
</tr>
<tr>
<td>setComment</td>
<td>getComment</td>
</tr>
<tr>
<td>getComment</td>
<td>setOwn</td>
</tr>
<tr>
<td>setOwn</td>
<td>getOwn</td>
</tr>
<tr>
<td>getOwn</td>
<td>print</td>
</tr>
</tbody>
</table>

**top half shows fields**

**bottom half shows methods**

DoME object model
public class CD {
    private String title;
    private String artist;
    private String comment;

    CD(String theTitle, String theArtist) {
        title = theTitle;
        artist = theArtist;
        comment = " ";
    }

    void setComment(String newComment) {
        ... 
    }

    String getComment() {
        ... 
    }

    void print() {
        ... 
    }

    ... 
}
public class Video {
    private String title;
    private String director;
    private String comment;

    Video(String theTitle, String theDirect) {
        title = theTitle;
        director = theDirect;
        comment = " ";
    }

    void setComment(String newComment) {
        ...}

    String getComment() {
        ...}

    void print() {
        ...
    }
}

class Database {
    private ArrayList cds;
    private ArrayList videos;
    ...

    public void list() {
        for(Iterator iter = cds.iterator(); iter.hasNext(); ) {
            CD cd = (CD)iter.next();
            cd.print();
            System.out.println(); // empty line between items
        }

        for(Iterator iter = videos.iterator(); iter.hasNext(); ) {
            Video video = (Video)iter.next();
            video.print();
            System.out.println(); // empty line between items
        }
    }
}
Critique of DoME

• code duplication
  – CD and Video classes very similar (large part are identical)
  – makes maintenance difficult/more work
  – introduces danger of bugs through incorrect maintenance

• code duplication also in Database class

Using inheritance
Using inheritance

• define one **superclass** : Item
• define **subclasses** for Video and CD
• the superclass defines common attributes
• the subclasses **inherit** the superclass members
• the subclasses add own members

Inheritance hierarchies
Inheritance in Java

public class Item
{
    // constructors and methods omitted.
}

public class CD extends Item
{
    // constructors and methods omitted.
}

public class Video extends Item
{
    // constructors and methods omitted.
}

Superclass

public class Item
{
    private String title;
    private int playingTime;
    private boolean gotIt;
    private String comment;

    // constructors and methods omitted.
}
Subclasses

public class CD extends Item
{
    private String artist;
    private int numberOfTracks;
    // constructors and methods omitted.
}

public class Video extends Item
{
    private String director;
    // constructors and methods omitted.
}

Inheritance and constructors

public class Item
{
    private String title;
    private int playingTime;
    private boolean gotIt;
    private String comment;

    /**
     * Initialise the fields of the item.
     */
    public Item(String theTitle, int time)
    {
        title = theTitle;
        playingTime = time;
        gotIt = false;
        comment = "";
    }

    // methods omitted
}
public class CD extends Item
{
    private String artist;
    private int numberOfTracks;

    /**
     * Constructor for objects of class CD
     */
    public CD(String theTitle, String theArtist,
              int tracks, int time)
    {
        super(theTitle, time);
        artist = theArtist;
        numberOfTracks = tracks;
    }

    // methods omitted
}

Superclass constructor call

- Subclass constructors must always contain a 'super' call.
- If none is written, the compiler inserts one (without parameters)
  - works only, if the superclass has a constructor without parameters
- Must be the first statement in the subclass constructor.
Adding more item types

Deeper hierarchies

Inheritance is transitive
Review (so far)

Inheritance (so far) helps with:
• Avoiding code duplication
• Code reuse
• Easier maintenance
• Extendibility

```java
public class Database {
    private ArrayList items;

    /**
     * Construct an empty Database.
     */
    public Database()
    {
        items = new ArrayList();
    }

    /**
     * Add an item to the database.
     */
    public void addItem(Item theItem)
    {
        items.add(theItem);
    }
    ...
}
```

New Database source code avoids code duplication in client!
New Database source code

```java
/**
 * Print a list of all currently stored CDs and videos to the text terminal.
 */
public void list()
{
    for(Iterator iter = items.iterator(); iter.hasNext(); ) {
        Item item = (Item)iter.next();
        item.print();
        System.out.println();   // empty line between items
    }
}
```

Subtyping

First, we had:
```java
public void addCD(CD theCD)
public void addVideo(Video theVideo)
```

Now, we have:
```java
public void addItem(Item theItem)
```

We call this method with:
```java
Video myVideo = new Video(...);
database.addItem(myVideo);
```
Subclasses and subtyping

- Classes define types.
- Subclasses define subtypes.
- Objects of subclasses can be used where objects of supertypes are required.
  (This is called substitution.)

Subtyping and assignment

```
Vehicle v1 = new Vehicle();
Vehicle v2 = new Car();
Vehicle v3 = new Bicycle();
```

subclass objects may be assigned to superclass variables
Subtyping and parameter passing

```java
class Database {
    public void addItem(Item theItem) {
        ...
    }
}

Video video = new Video(...);
CD cd = new CD(...);

database.addItem(video);
database.addItem(cd);
```

Object diagram

![Object diagram](image-url)
Abstraction

- Focus on commonalities among objects in system
- “is-a” vs. “has-a”
  - “is-a”
    - Inheritance
    - Upperclass represents an abstraction (more generic concept) of a lower class
    - subclass object treated as superclass object
    - Example: Car is a vehicle
      - Vehicle properties/behaviors also car properties/behaviors
  - “has-a”
    - Composition
    - Object contains one or more objects of other classes as members
    - Example: Car has a steering wheel
protected Members

- **protected** access
  - Intermediate level of protection between **public** and **private**
  - **protected** members of a superclass are accessible to
    - superclass members
    - subclass members
    - Class members in the same package
  - Subclass access superclass member
    - Keyword **super** and a dot (.)
  - Protected does not allow access to a class’s methods to clients other than subclasses.
  - declare superclass instance variables private
    - superclass implementation can change without affecting subclasses

- private members are not inherited by subclasses
- public members are inherited by subclasses

Relationship between Superclasses and Subclasses

- Superclass and subclass relationship
  - Example: Point/circle inheritance hierarchy
    - Point
      - x-y coordinate pair
    - Circle
      - x-y coordinate pair
      - Radius

- Constructors are not inherited
  - Direct superclass’s constructor is called implicitly (with no arguments) or explicitly (possibly with several arguments).
// Fig. 9.4: Point.java
// Point class declaration represents an x-y coordinate pair.

public class Point {
    private int x; // x part of coordinate pair
    private int y; // y part of coordinate pair

    // no-argument constructor
    public Point() {
        // implicit call to Object constructor occurs here
    }

    // constructor
    public Point(int xValue, int yValue) {
        // implicit call to Object constructor occurs here
        x = xValue; // no need for validation
        y = yValue; // no need for validation
    }

    // set x in coordinate pair
    public void setX(int xValue) {
        x = xValue; // no need for validation
    }

    // return x from coordinate pair
    public int getX() {
        return x;
    }

    // set y in coordinate pair
    public void setY(int yValue) {
        y = yValue; // no need for validation
    }

    // return y from coordinate pair
    public int getY() {
        return y;
    }

    // return String representation of Point object
    public String toString() {
        return "[" + x + ", " + y + "]";
    }
}

// end class Point
// Fig. 9.5: PointTest.java
// Testing class Point.
import javax.swing.JOptionPane;

public class PointTest {
    public static void main( String[] args ) {
        Point point = new Point( 72, 115 );  // create Point object

        // get point coordinates
        String output = "X coordinate is " + point.getX() + 
                         "\nY coordinate is " + point.getY();

        point.setX( 10 );  // set x-coordinate
        point.setY( 20 );  // set y-coordinate

        // get String representation of new point value
        output += "\nThe new location of point is " + point;

        JOptionPane.showMessageDialog( null, output ); // display output
        System.exit( 0 );
    } // end main
} // end class PointTest

// Fig. 9.6: Circle.java
// Circle class contains x-y coordinate pair and radius.
public class Circle {
    private int x;          // x-coordinate of Circle's center
    private int y;          // y-coordinate of Circle's center
    private double radius;  // Circle's radius

    // no-argument constructor
    public Circle() {
        // implicit call to Object constructor occurs here
    }

    // constructor
    public Circle( int xValue, int yValue, double radiusValue ) {
        // implicit call to Object constructor occurs here
        x = xValue;  // no need for validation
        y = yValue;  // no need for validation
        setRadius( radiusValue );
    }

    // set x in coordinate pair
    public void setX( int xValue ) {
        x = xValue;  // no need for validation
    }

    setRadius( radiusValue );
}

Note code similar to Point code.
// return x from coordinate pair
public int getX()
{
    return x;
}

// set y in coordinate pair
public void setY( int yValue )
{
    y = yValue; // no need for validation
}

// return y from coordinate pair
public int getY()
{
    return y;
}

// set radius
public void setRadius( double radiusValue )
{
    radius = ( radiusValue < 0.0 ? 0.0 : radiusValue );
}

// return radius
public double getRadius()
{
    return radius;
}

// calculate and return diameter
public double getDiameter()
{
    return 2 * radius;
}

// calculate and return circumference
public double getCircumference()
{
    return Math.PI * getDiameter();
}

// calculate and return area
public double getArea()
{
    return Math.PI * radius * radius;
}

// return String representation of Circle object
public String toString()
{
    return "Center = [" + x + ", " + y + "]; Radius = " + radius;
}

} // end class Circle
Create Circle object.

Lines 17-19
Use set methods to modify private instance variable.

Explicitly call circle's toString method to modify private instance variable.

Use get methods to obtain circle's diameter, circumference and area.

Use get methods to obtain circle's diameter, circumference and area.
Class Circle2 extends class Point.

Maintain private instance variable radius.

Attempting to access superclass Point's private instance variables x and y results in syntax errors.

Lines 17-18

Attempting to access superclass Point's private instance variables x and y results in syntax errors.

Attempting to access superclass Point's private instance variables x and y results in syntax errors.
Circle2.java:17: x has private access in Point
   x = xValue; // not allowed: x private in Point
   ^
Circle2.java:18: y has private access in Point
   y = yValue; // not allowed: y private in Point
   ^
Circle2.java:56: x has private access in Point
   return "Center = [" + x + ", " + y + "] ; Radius = " + radius;
   ^
Circle2.java:56: y has private access in Point
   return "Center = [" + x + ", " + y + "] ; Radius = " + radius;
   ^
4 errors

Attempting to access superclass Point’s private instance variables x and y results in syntax errors.

// Fig. 9.9: Point2.java
// Point2 class declaration represents an x-y coordinate pair.

public class Point2 {
   protected int x;  // x part of coordinate pair
   protected int y;  // y part of coordinate pair

   // no-argument constructor
   public Point2()
   {
      // implicit call to Object constructor occurs here
   }

   // constructor
   public Point2( int xValue, int yValue )
   {
      // implicit call to Object constructor occurs here
      x = xValue; // no need for validation
      y = yValue; // no need for validation
   }

   // set x in coordinate pair
   public void setX( int xValue )
   {
      x = xValue; // no need for validation
   }

   // set y in coordinate pair
   public void setY( int yValue )
   {
      y = yValue; // no need for validation
   }

   // get x in coordinate pair
   public int getX()
   {
      return x;
   }

   // get y in coordinate pair
   public int getY()
   {
      return y;
   }

   // print the x-y coordinate pair
   public String toString()
   {
      return "(" + x + ", " + y + ");"
   }
}

Maintain x- and y-coordinates as protected instance variables, accessible to subclasses.
// return x from coordinate pair
public int getX()
{
    return x;
}

// set y in coordinate pair
public void setY( int yValue )
{
    y = yValue;  // no need for validation
}

// return y from coordinate pair
public int getY()
{
    return y;
}

// return String representation of Point2 object
public String toString()
{
    return "[" + x + ", " + y + "]";
}

} // end class Point2

public class Circle3 extends Point2 {

    private double radius;  // Circle3's radius

    // no-argument constructor
    public Circle3()
    {
    // implicit call to Point2 constructor occurs here
    }

    // constructor
    public Circle3( int xValue, int yValue, double radiusValue )
    {
    // implicit call to Point2 constructor occurs here
    x = xValue; // no need for validation
    y = yValue;  // no need for validation
    setRadius( radiusValue );
    }

    // set radius
    public void setRadius( double radiusValue )
    {
        radius = ( radiusValue < 0.0 ? 0.0 : radiusValue );
    }

    // Fig. 9.10: Circle3.java
    // Circle3 class inherits from Point2 and has access to Point2's protected members x and y.

    public class Circle3 extends Point2 {  // Circle3 inherits from Point2
        private double radius;  // Circle3's radius

        // no-argument constructor
        public Circle3()
        {
        // implicit call to Point2 constructor occurs here
        }

        // constructor
        public Circle3( int xValue, int yValue, double radiusValue )
        {
        // implicit call to Point2 constructor occurs here
        x = xValue; // no need for validation
        y = yValue;  // no need for validation
        setRadius( radiusValue );
        }

        // set radius
        public void setRadius( double radiusValue )
        {
            radius = ( radiusValue < 0.0 ? 0.0 : radiusValue );
        }
    }

    Class Circle3 inherits from Point2.
    Maintain private instance variables radius.

    Modifier calls superclass's default constructor.
    Modify inherited instance variables x and y, declared protected in superclass Point2.

    Lines 11 and 17
    Implicitly call superclass's default constructor.

    Lines 18-19
    Modify inherited instance variables x and y, declared protected in superclass Point2.
Circle3.java

Line 56
Access inherited instance variables \( x \) and \( y \), declared protected in superclass \CodePoint\Point2\CodePoint.

Outline

Circletest3.java

Line 11
Create Circle3 object.

Use inherited get methods to access inherited protected instance variables.

Lines 14-15
Use inherited get methods to access inherited protected instance variables.

Lines 18-19
Use inherited set methods to modify inherited protected data \( x \) and \( y \).

Line 20
Use Circle3 set method to modify private data \( radius \).
// format floating-point values with 2 digits of precision
DecimalFormat twoDigits = new DecimalFormat( "0.00" );

// get Circle's diameter
output += "\nDiameter is " +
twoDigits.format( circle.getDiameter() );

// get Circle's circumference
output += "\nCircumference is " +
twoDigits.format( circle.getCircumference() );

// get Circle's area
output += "\nArea is " + twoDigits.format( circle.getArea() );

JOptionPane.showMessageDialog( null, output ); // display output
System.exit( 0 );

} // end method main

} // end class CircleTest3

Relationship between Superclasses and Subclasses (Cont.)

• Using protected instance variables
  – Advantages
    • subclasses can modify values directly
    • Slight increase in performance
      – Avoid set/get function call overhead
  – Disadvantages
    • No validity checking
      – subclass can assign illegal value
    • Implementation dependent
      – subclass methods more likely dependent on superclass
      implementation
      – superclass implementation changes may result in subclass
      modifications
        • Fragile (brittle) software
Relationship between Superclasses and Subclasses (Cont.)

- Override a method with a more restricted access modifier → syntax error
- A public method of the superclass cannot become a protected or private method in the subclass.
- Declare superclass instance variables private (as opposed to protected), thus superclass implementation can change without affecting subclass implementation.
- instead of protected instance variables:
  - non-private methods for accessing private variables

// Fig. 9.12: Point3.java
// Point class declaration represents an x-y coordinate pair.

```java
public class Point3 {
    private int x; // x part of coordinate pair
    private int y; // y part of coordinate pair

    // no-argument constructor
    public Point3() {
        // implicit call to Object constructor occurs here
    }

    // constructor
    public Point3( int xValue, int yValue ) {
        // implicit call to Object constructor occurs here
        x = xValue; // no need for validation
        y = yValue; // no need for validation
    }

    // set x in coordinate pair
    public void setX( int xValue ) {
        x = xValue; // no need for validation
    }
}
```

Better software-engineering practice: private over protected when possible.
// return x from coordinate pair
public int getX()
{
    return x;
}

// set y in coordinate pair
public void setY( int yValue )
{
    y = yValue;  // no need for validation
}

// return y from coordinate pair
public int getY()
{
    return y;
}

// return String representation of Point3 object
public String toString()
{
    return "[" + getX() + ", " + getY() + "]";
}

} // end class Point3

// Fig. 9.13: Circle4.java
// Circle4 class inherits from Point3 and accesses Point3's public methods
public class Circle4 extends Point3 {
    
    private double radius;  // Circle4's radius

    // no-argument constructor
    public Circle4()
    {
        // implicit call to Point3 constructor occurs here
    }

    // constructor
    public Circle4( int xValue, int yValue, double radiusValue )
    {
        super( xValue, yValue );  // call Point3 constructor explicitly
        setRadius( radiusValue );
    }

    // set radius
    public void setRadius( double radiusValue )
    {
        radius = ( radiusValue < 0.0 ? 0.0 : radiusValue );
    }

} // end class Circle4

Outline

Point3.java
Line 49
Invoke public methods to access private instance variables.

Class Circle4 inherits from class Point3.
Maintain private instance variable radius.

Circle4.java
Line 5
Class Circle4 inherits from class Point3.

Line 7
Maintain private instance variable radius.
Circle4.java

Line 37, 49 and 55
Invoke method `getRadius` rather than directly accessing instance variable `radius`.

Lines 53-56
Redefine class `Point3`'s method `toString`.

CircleTest4.java

Line 11
Create `Circle4` object.

Use inherited get methods to access inherited private variable radius.

Lines 14 and 15
Use `Circle4` get method to access private instance variable radius.

Lines 18-19
Use inherited seta methods to modify inherited private variable radius.

Line 20
Use `Circle4` set method to modify private instance variable radius.
// format floating-point values with 2 digits of precision
DecimalFormat twoDigits = new DecimalFormat("0.00");

// get Circle's diameter
output += "Diameter is " +
twoDigits.format(circle.getDiameter());

// get Circle's circumference
output += "Circumference is " +
twoDigits.format(circle.getCircumference());

// get Circle's area
output += "Area is " + twoDigits.format(circle.getArea());

JOptionPane.showMessageDialog(null, output); // display output
System.exit(0);

} // end main

} // end class CircleTest4

Case Study: Three-Level Inheritance Hierarchy

- Three level point/circle/cylinder hierarchy
  - Point
    - x-y coordinate pair
  - Circle
    - x-y coordinate pair
    - Radius
  - Cylinder
    - x-y coordinate pair
    - Radius
    - Height
Cylinder.java

Class Cylinder extends class Circle4.

Line 5
Maintain private instance variable height.

Redefine superclass Circle4's method getArea to return Cylinder surface area.

Invoke superclass Circle4's method toString.

Redefine superclass Circle4's method getArea to return Cylinder surface area.

Lines 46-49
Redefine class Circle4's method toString.

Line 48
Invoke superclass Circle4's method toString.

Outline
public class CylinderTest {
    public static void main( String[] args ) {
        // create Cylinder object
        Cylinder cylinder = new Cylinder( 12, 23, 2.5, 5.7 );
        // get Cylinder's initial x-y coordinates, radius and height
        String output = "X coordinate is " + cylinder.getX() +
            "Y coordinate is " + cylinder.getY() +
            "Radius is " + cylinder.getRadius() +
            "Height is " + cylinder.getHeight();
        cylinder.setX( 35 );          // set new x-coordinate
        cylinder.setY( 20 );          // set new y-coordinate
        cylinder.setRadius( 4.25 );   // set new radius
        cylinder.setHeight( 10.75 );  // set new height
        // get String representation of new cylinder value
        output +=
            "The new location, radius and height of cylinder are\n" +
            cylinder.toString();
        // format floating-point values with 2 digits of precision
        DecimalFormat twoDigits = new DecimalFormat( "0.00" );
        // get Cylinder's diameter
        output += "Diameter is " +
            twoDigits.format( cylinder.getDiameter() );
        // get Cylinder's circumference
        output += "Circumference is " +
            twoDigits.format( cylinder.getCircumference() );
        // get Cylinder's area
        output += "Area is " +
            twoDigits.format( cylinder.getArea() );
        // get Cylinder's volume
        output += "Volume is " +
            twoDigits.format( cylinder.getVolume() );
        JOptionPane.showMessageDialog( null, output ); // display output
        System.exit( 0 );
    } // end main
} // end class CylinderTest
Constructors and Finalizers in Subclasses

• Instantiating subclass object
  – Chain of constructor calls
    • subclass constructor invokes superclass constructor
      – Implicitly or explicitly
      – subclass constructor immediately calls the superclass constructor
        (explicitly, via super or implicitly)
  • Base of inheritance hierarchy
    – Last constructor called in chain is Object’s constructor
    – Original subclass constructor’s body finishes executing last
    – Example: Point3/Circle4/Cylinder hierarchy
     • Point3 constructor called second last (last is Object constructor)
     • Point3 constructor’s body finishes execution second (first is Object constructor’s body)
     • Circle4 and Cylinder constructor’s body finishes execution third and fourth, respectively

9.6 Constructors and Destructors in Derived Classes

• Garbage collecting subclass object
  – Chain of finalize method calls
    • Reverse order of constructor chain
    • Finalizer of subclass called first
    • Finalizer of next superclass up hierarchy next
      – Continue up hierarchy until final superreached
        • After final superclass (Object) finalizer, object removed from memory
// Fig. 9.17: Point.java
// Point class declaration represents an x-y coordinate pair.

public class Point {
    private int x;  // x part of coordinate pair
    private int y;  // y part of coordinate pair

    // no-argument constructor
    public Point() {
        // implicit call to Object constructor occurs here
        System.out.println( "Point no-argument constructor: " + this );
    }

    // constructor
    public Point( int xValue, int yValue ) {
        // implicit call to Object constructor occurs here
        x = xValue;  // no need for validation
        y = yValue;  // no need for validation
        System.out.println( "Point constructor: " + this );
    }

    // finalizer
    protected void finalize() {
        System.out.println( "Point finalizer: " + this );
    }

    // set x in coordinate pair
    public void setX( int xValue ) {
        x = xValue;  // no need for validation
    }

    // return x from coordinate pair
    public int getX() {
        return x;
    }

    // set y in coordinate pair
    public void setY( int yValue ) {
        y = yValue;  // no need for validation
    }

    // return y from coordinate pair
    public int getY() {
        return y;
    }

    // return String representation of Point4 object
    public String toString() {
        return "[" + getX() + ", " + getY() + "]";
    }
}

// end class Point
// Fig. 9.18: Circle.java
// Circle5 class declaration.

public class Circle extends Point {
    private double radius;  // Circle's radius

    // no-argument constructor
    public Circle() {
        // implicit call to Point constructor occurs here
        System.out.println( "Circle no-argument constructor: " + this );
    }

    // constructor
    public Circle( int xValue, int yValue, double radiusValue )
    {
        super( xValue, yValue );  // call Point constructor
        setRadius( radiusValue );
        System.out.println( "Circle constructor: " + this );
    }

    // finalizer
    protected void finalize() {
        System.out.println( "Circle finalizer: " + this );
        super.finalize();  // call superclass finalize method
    }

    // set radius
    public void setRadius( double radiusValue )
    {
        radius = ( radiusValue < 0.0 ? 0.0 : radiusValue );
    }

    // return radius
    public double getRadius() {
        return radius;
    }

    // calculate and return diameter
    public double getDiameter() {
        return 2 * getRadius();
    }

    // calculate and return circumference
    public double getCircumference() {
        return Math.PI * getDiameter();
    }
}
Circle.java

```java
public double getArea()
{
    return Math.PI * getRadius() * getRadius();
}

public String toString()
{
    return "Center = " + super.toString() + "; Radius = " + getRadius();
}
}
```

ConstructorFinalizerTest.java

```java
public class ConstructorFinalizerTest {
    public static void main(String args[])
    {
        Point point;
        Circle circle1, circle2;

        point = new Point(11, 22);
        System.out.println();
        circle1 = new Circle(72, 29, 4.5);
        System.out.println();
        circle2 = new Circle(5, 7, 10.67);
        System.out.println();
        point = null; // mark for garbage collection
        circle1 = null; // mark for garbage collection
        circle2 = null; // mark for garbage collection
        System.out.println();
    }
}
```
Summary

- Inheritance allows the definition of classes as extensions of other classes.
- Inheritance
  - avoids code duplication
  - allows code reuse
  - simplifies the code
  - simplifies maintenance and extending
Part 12
Object Oriented Programming: Polymorphism

- polymorphism
- abstract versus concrete classes
- interfaces
- inner classes

Polymorphism: Motivation

- Assumption: all classes have method getName();
- superclass variable can be assigned reference to subclass object.
- superclass variable can invoke getName method of subclass objects
- execution time environment polymorphically chooses correct subclass version of the method based on the type of the reference stored in the superclass variable.
Introduction: Polymorphism (1)

- A variable may reference objects of different classes.
- Objects of a subclass $A$ can be assigned to objects of a superclass (assignment compatible) or to objects of class $A$.
- A superclass object is not a subclass object.

```
Person p;   // p may refer to objects of class Person or Student.
Student s;
...
p = s;     // legal; every student is a Person
s = p;     // illegal; not every person is a student
```

Introduction: Polymorphism (2)

- An object of a derived class $B$ has the types of all superclasses of $B$.
  e.g.: $s$ has type $\textbf{Student}$ and type $\textbf{Person}$

```
Student s = new Student();
boolean test1 = s instanceof Student;   // yields true
boolean test2 = s instanceof Person;    // yields true
```
Assignment Compatibility

- Variables with type B can be assigned to variables of type B or type A where A is a superclass of B.

class Person { ...
class Student extends Person { ...
class Kurs {
  ...
  void register(Person p) { ...
}
...
Person p = new Person();
Student s = new Student();
s = p; // Error! (incompatible)
p = s; // compatible
Kurs kurs = new Kurs();
kurs.register(s); // compatible

Inheritance and Cast Operator (1)

- If a variable a of type A references an object of a subclass B, then a can be assigned to b of type B by using an explicit type cast (downcasting).

class A { ...
class B extends A { ...

A a;
B b = new B();
a = b;
b = (B) a; // cast type of a to B
Inheritance and Cast Operator (2)

Vehicle f;
Car a;            // Car is a subclass of Vehicle
a = new Car();
f = a;            // f now becomes a car
a = (Car) f;     // correct because f is a car

Vehicle f;
Car a;
Bike b;           // Bike is a subclass of Vehicle
a = new Car();
f = a;            // f now becomes a car
b = (Bike) a;     // error detected during compilation
                    // no subclass-superclass relation between a and b
b = (Bike) f;     // error detected during runtime
                    // f is not a bike at this point

// Fig. 10.1: HierarchyRelationshipTest1.java
// Assigning superclass and subclass references to superclass- and
// subclass-type variables.
import javax.swing.JOptionPane;

public class HierarchyRelationshipTest1 {
    public static void main( String[] args )
    {
        // assign superclass reference to superclass-type variable
        Point3 point = new Point3( 30, 50 );

        // assign subclass reference to subclass-type variable
        Circle4 circle = new Circle4( 120, 89, 2.7 );

        // invoke toString on superclass object using superclass variable
        String output = "Call Point3's toString with superclass reference to superclass object: \n" + point.toString();

        // invoke toString on subclass object using subclass variable
        output += "Call Circle4's toString with subclass reference to subclass object: \n" + circle.toString();
    }
}
Using Superclass References with Subclass-Type Variables

- Previous example
  - Assigned subclass reference to superclass-type variable
    - Circle “is a” Point

- Assign superclass reference to subclass-type variable
  - Compiler error
    - No “is a” relationship
    - Point is not a Circle
    - Circle has data/methods that Point does not
      - setRadius (declared in Circle) not declared in Point
  - Cast superclass references to subclass references
    - Called downcasting
    - Invoke subclass functionality
Assigning superclass reference to subclass-type variable causes compiler error.

### Subclass Method Calls via Superclass-Type variables

- Call a subclass method with superclass reference
  - Compiler error
    - Subclass methods are not superclass methods
- Works if superclass contains definition of the same method as in subclass
// Fig. 10.3: HierarchyRelationshipTest3.java
// Attempting to invoke subclass-only member methods through
// a superclass reference.

class HierarchyRelationshipTest3 {
    public static void main(String[] args) {
        Point3 point;
        Circle4 circle = new Circle4(120, 89, 2.7);
        point = circle; // aim superclass reference at subclass object

        // invoke superclass (Point3) methods on subclass
        // (Circle4) object through superclass reference
        int x = point.getX(); // invoke getX of class Point3
        int y = point.getY(); // invoke getY of class Point3
        point.setX(10); // invoke setX of class Point3
        point.setY(20); // invoke setY of class Point3
        point.toString(); // invoke toString of class Circle4

        // attempt to invoke subclass-only (Circle4) methods on subclass object through superclass (Point3) reference
        double radius = point.getRadius();
        point.setRadius(33.33);
        double diameter = point.getDiameter();
        double circumference = point.getCircumference();
        double area = point.getArea();
    }
}

A variable can be used to invoke only methods that are members of that variable's type.
This can be checked by the compiler at compilation time.
Abstract Classes

• Abstract classes
  – are used for the representation of abstract concepts (vehicle, planet, etc.) in Java.
  – are used for closely related classes (vehicle, car, truck, etc.)
  – are superclasses (called abstract superclasses) in inheritance hierarchies
  – cannot be instantiated
  – Incomplete (only partially implemented)
    • subclasses fill in "missing pieces"

• Concrete classes
  – Can be instantiated
  – Implement every method they declare
  – Provide specifics

```java
public abstract class Vehicle {
    
}
```
Abstract Methods

- declared with the keyword `abstract`
- defines only the `interface` (types and names of parameters and return value)
- no implementation (body)

```java
abstract class Food { // abstract class
    abstract double nutrition (double amount); // abstract method
}
```

Abstract Classes and Methods (1)

- Every class with at least one abstract method must be declared to be `abstract` (abstract class ...)
- Abstract classes may contain – besides abstract methods – arbitrary fields, non-abstract methods and constructors.

```java
abstract class Shape { // abstract Class
    static int nrOfShapes; // class variable
    Point origin; // instance variable
    Shape() {nrOfShapes++;} // constructor
    abstract double area(); // abstract method
}
```
Abstract Classes and Methods (2)

A subclass B of an abstract class A can be instantiated if B implements all abstract methods of A.

```java
abstract class Shape { // abstract class
    static int nrOfShapes; // class variable
    Point origin; // instance variable
    Shape() {nrOfShapes++; } // constructor
    abstract double area(); // abstract method
}
class Circle extends Shape { // subclass of Shape
double r;
double area() { return r*r*3.14;} }
...
Circle c = new Circle();
Shape s = c;
System.out.println(s.area());
```

Abstract Classes and Methods (3)

If a subclass B does not implement all abstract methods of an abstract class A, then B is an abstract class as well.

```java
abstract class Shape { // abstract class
    static int nrOfShapes; // class variable
    Point origin; // instance variable
    Shape() {nrOfShapes++; } // constructor
    abstract double area(); // abstract method
}
abstract class ColoredShape extends Shape { // abstract subclass of Shape
    Color c;
    void setColor(Color newCol) { c = newCol; }
}
```
Case Study: Inheriting Interface and Implementation

- Make abstract superclass Shape
  - Abstract method
    - getName
      - Default implementation does not make sense
  - Methods may be overridden
    - getArea, getVolume
      - Default implementations return 0.0
    - If not overridden, uses superclass default implementation
  - Subclasses Point, Circle, Cylinder

10.5 Case Study: Inheriting Interface and Implementation

Fig. 10.4 Shape hierarchy class diagram.
Case Study: Inheriting Interface and Implementation

<table>
<thead>
<tr>
<th></th>
<th>getArea</th>
<th>getVolume</th>
<th>getName</th>
<th>print</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shape</td>
<td>0.0</td>
<td>0.0</td>
<td>abstract</td>
<td>default Object implement.</td>
</tr>
<tr>
<td>Point</td>
<td>0.0</td>
<td>0.0</td>
<td>&quot;Point&quot;</td>
<td>[x,y]</td>
</tr>
<tr>
<td>Circle</td>
<td>pr^2</td>
<td>0.0</td>
<td>&quot;Circle&quot;</td>
<td>center=[x,y]; radius=r</td>
</tr>
<tr>
<td>Cylinder</td>
<td>2pr^2+2prh</td>
<td>pr^2h</td>
<td>&quot;Cylinder&quot;</td>
<td>center=[x,y]; radius=r; height=h</td>
</tr>
</tbody>
</table>

Polymorphic interface for the Shape hierarchy classes.

```
// Fig. 10.6: Shape.java
// Shape abstract-superclass declaration.
public abstract class Shape extends Object {
    // return area of shape; 0.0 by default
    public double getArea() {
        return 0.0;
    }
    // return volume of shape; 0.0 by default
    public double getVolume() {
        return 0.0;
    }
    // abstract method, overridden by subclasses
    public abstract String getName();
} // end abstract class Shape
```
public class Point extends Shape {
  private int x; // x part of coordinate pair
  private int y; // y part of coordinate pair

  // no-argument constructor; x and y default to 0
  public Point() {
    // implicit call to Object constructor occurs here
  }

  // constructor
  public Point( int xValue, int yValue ) {
    // implicit call to Object constructor occurs here
    x = xValue; // no need for validation
    y = yValue; // no need for validation
  }

  // set x in coordinate pair
  public void setX( int xValue ) {
    x = xValue; // no need for validation
  }

  // return x from coordinate pair
  public int getX() {
    return x;
  }

  // set y in coordinate pair
  public void setY( int yValue ) {
    y = yValue; // no need for validation
  }

  // return y from coordinate pair
  public int getY() {
    return y;
  }

  // override abstract method getName to return "Point"
  public String getName() {
    return "Point";
  }

  // override toString to return String representation of Point
  public String toString() {
    return "[" + getX() + ", " + getY() + "]";
  }
}

// end class Point
// Fig. 10.8: Circle.java
// Circle class inherits from Point.

class Circle extends Point {
    private double radius;  // Circle's radius

    // no-argument constructor; radius defaults to 0.0
    public Circle() {
        // implicit call to Point constructor occurs here
    }

    // constructor
    public Circle( int x, int y, double radiusValue )
    {
        super( x, y );  // call Point constructor
        setRadius( radiusValue );
    }

    // set radius
    public void setRadius( double radiusValue )
    {
        radius = ( radiusValue < 0.0 ? 0.0 : radiusValue );
    }

    // return radius
    public double getRadius() {
        return radius;
    }

    // calculate and return diameter
    public double getDiameter() {
        return 2 * getRadius();
    }

    // calculate and return circumference
    public double getCircumference() {
        return Math.PI * getDiameter();
    }

    // override method to return circle area
    public double getArea() {
        return Math.PI * getRadius() * getRadius();
    }
}
Circle.java

Lines 51-54
Override abstract method getName.

50    // override abstract method getName to return "Circle"
51    public String getName() {
52    return "Circle";
53    }
54
55    // override toString to return String representation of Circle
56    public String toString() {
57    return "Center = " + super.toString() + "; Radius = " + getRadius();
58    }
59
60    } // end class Circle

Cylinder.java

1 // Fig. 10.9: Cylinder.java
2 // Cylinder class inherits from Circle.
3
4    public class Cylinder extends Circle {
5    private double height;  // Cylinder's height
6
7    // no-argument constructor; height defaults to 0.0
8    public Cylinder() {
9    // implicit call to Circle constructor occurs here
10    }
11
12    // constructor
13    public Cylinder( int x, int y, double radius, double heightValue ) {
14    super( x, y, radius );  // call Circle constructor
15    setHeight( heightValue );
16    }
17
18    // set Cylinder's height
19    public void setHeight( double heightValue ) {
20    height = ( heightValue < 0.0 ? 0.0 : heightValue );
21    }
22
23    } // end class Cylinder
Override method `getArea` to return cylinder area

Lines 33-36
Override method `getVolume` to return cylinder volume

Lines 39-42
Override abstract method `getName` to return "Cylinder"

Lines 45-48
Override abstract method `getName`
// Fig. 10.10: AbstractInheritanceTest.java
// Driver for shape, point, circle, cylinder hierarchy.
import java.text.DecimalFormat;
import javax.swing.JOptionPane;

public class AbstractInheritanceTest {
    public static void main( String args[] )
    {
        // set floating-point number format
        DecimalFormat twoDigits = new DecimalFormat( "0.00" );
        // create Point, Circle and Cylinder objects
        Point point = new Point( 7, 11 );
        Circle circle = new Circle( 22, 8, 3.5 );
        Cylinder cylinder = new Cylinder( 20, 30, 3.3, 10.75 );
        // obtain name and string representation of each object
        String output = point.getName() +": " + point + "\n" +
                        circle.getName() +": " + circle + "\n" +
                        cylinder.getName() +": " + cylinder + "\n";
        Shape arrayOfShapes[] = new Shape[3];  // create Shape array
        // aim arrayOfShapes[0] at subclass Point object
        arrayOfShapes[0] = point;
        // aim arrayOfShapes[1] at subclass Circle object
        arrayOfShapes[1] = circle;
        // aim arrayOfShapes[2] at subclass Cylinder object
        // loop through arrayOfShapes to get name, string representation, area and volume of every shape in array
        for ( int i = 0; i < arrayOfShapes.length; i++ ) {
            output += "\n\n" + arrayOfShapes[i].getName() +": " +
                       arrayOfShapes[i].toString() + "\nArea = " +
                       twoDigits.format( arrayOfShapes[i].getArea() ) +
                       "\nVolume = " +
                       twoDigits.format( arrayOfShapes[i].getVolume() );
        }
        JOptionPane.showMessageDialog( null, output );  // display output
        System.exit( 0 );
    } // end main
} // end class AbstractInheritanceTest

late binding or dynamic binding: All method calls to getName, toString, getArea and getVolume are resolved at execution time, based on the type of the object to which element arrayOfShapes[i] currently refers.
final Methods and Classes

- **final methods**
  - in superclasses cannot be overridden in subclasses
  - **private** methods are implicitly **final**
  - **static** methods are implicitly **final**
  - only non-static methods can be overridden in subclasses.

- **final classes**
  - Cannot be superclasses
  - Methods in **final** classes are implicitly **final**
  - e.g., class **String**
  - final classes typically for security reasons
Case Study: Payroll System Using Polymorphism

• Create a payroll program
  – Use abstract methods and polymorphism

• Problem statement
  – 4 types of employees, paid weekly
    • Salaried (fixed salary, no matter the hours)
    • Hourly (overtime [>40 hours] pays time and a half)
    • Commission (paid percentage of sales)
    • Base-plus-commission (base salary + percentage of sales)
      – Boss wants to raise pay by 10%

Case Study: Payroll System Using Polymorphism

• Superclass Employee
  – Abstract method `earnings` (returns pay)
    • abstract because need to know employee type
    • Cannot calculate for generic employee
  – Other classes extend Employee
public abstract class Employee {
    private String firstName;
    private String lastName;
    private String socialSecurityNumber;

    public Employee(String first, String last, String ssn) {
        firstName = first;
        lastName = last;
        socialSecurityNumber = ssn;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String first) {
        firstName = first;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String last) {
        lastName = last;
    }

    public String getSocialSecurityNumber() {
        return socialSecurityNumber;
    }

    public void setSocialSecurityNumber(String number) {
        socialSecurityNumber = number; // should validate
    }
}
Abstract method overridden by subclasses.

```java
// return social security number
public String getSocialSecurityNumber() {
    return socialSecurityNumber;
}

// return String representation of Employee object
public String toString() {
    return getFirstName() + " " + getLastName() +
    "\nsocial security number: " + getSocialSecurityNumber();
}

// abstract method overridden by subclasses
public abstract double earnings();
```

Abstract method overridden by subclasses.

```java
// Fig. 10.13: SalariedEmployee.java
// SalariedEmployee class extends Employee.

public class SalariedEmployee extends Employee {
    private double weeklySalary;

    // constructor
    public SalariedEmployee( String first, String last, String socialSecurityNumber, double salary )
    {
        super( first, last, socialSecurityNumber );
        setWeeklySalary( salary );
    }

    // set salaried employee's salary
    public void setWeeklySalary( double salary )
    {
        weeklySalary = salary < 0.0 ? 0.0 : salary;
    }

    // return salaried employee's salary
    public double getWeeklySalary() {
        return weeklySalary;
    }
```
SalariedEmployee.java

Lines 29-32
Must implement abstract method earnings.

```java
public double earnings() {
    return getWeeklySalary();
}
```

HourlyEmployee.java

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

    // constructor
    public HourlyEmployee( String first, String last,
                           String socialSecurityNumber, double hourlyWage, double hoursWorked )
    {
        super( first, last, socialSecurityNumber );
        setWage( hourlyWage );
        setHours( hoursWorked );
    }

    // set hourly employee's wage
    public void setWage( double wageAmount )
    {
        wage = wageAmount < 0.0 ? 0.0 : wageAmount;
    }

    // return wage
    public double getWage()
    {
        return wage;
    }
```
// set hourly employee's hours worked
public void setHours( double hoursWorked )
{
    hours = ( hoursWorked >= 0.0 && hoursWorked <= 168.0 ) ?
        hoursWorked : 0.0;
}

// return hours worked
public double getHours()
{
    return hours;
}

// calculate hourly employee's pay;
// override abstract method earnings in Employee
public double earnings()
{
    if ( hours <= 40 ) // no overtime
        return wage * hours;
    else
        return 40 * wage + ( hours - 40 ) * wage * 1.5;
}

// return String representation of HourlyEmployee object
public String toString()
{
    return "\nhourly employee: " + super.toString();
}
}

// Fig. 10.15: CommissionEmployee.java
// CommissionEmployee class extends Employee.
public class CommissionEmployee extends Employee {
    private double grossSales; // gross weekly sales
    private double commissionRate; // commission percentage

    // constructor
    public CommissionEmployee( String first, String last,
        String socialSecurityNumber,
        double grossWeeklySales, double percent )
    {
        super( first, last, socialSecurityNumber );
        setGrossSales( grossWeeklySales );
        setCommissionRate( percent );
    }

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

    // return commission employee's rate
    public double getCommissionRate()
    {
        return commissionRate;
    }
}
public void setGrossSales( double sales )
{
    grossSales = sales < 0.0 ? 0.0 : sales;
}

public double getGrossSales()
{
    return grossSales;
}

public double earnings()
{
    return getCommissionRate() * getGrossSales();
}

public String toString()
{
    return super.toString();
}
// calculate base-salaried commission employee's earnings;
public double earnings()
{
    return getBaseSalary() + super.earnings();
}

// return String representation of BasePlusCommissionEmployee
public String toString()
{
    return "base-salaried commission employee: " +
    super.getFirstName() + " " + super.getLastName() +
    "social security number: " + super.getSocialSecurityNumber();
}

} // end class BasePlusCommissionEmployee

// Fig. 10.17: PayrollSystemTest.java
// Employee hierarchy test program.
import java.text.DecimalFormat;
import javax.swing.JOptionPane;

public class PayrollSystemTest {
    public static void main( String[] args )
    {
        DecimalFormat twoDigits = new DecimalFormat( "0.00" );

        // create Employee array
        Employee employees[] = new Employee[ 4 ];

        // initialize array with Employees
        employees[ 0 ] = new SalariedEmployee( "John", "Smith", 
            "111-11-1111", 800.00 );
        employees[ 1 ] = new CommissionEmployee( "Sue", "Jones", 
            "222-22-2222", 10000, .06 );
            "333-33-3333", 5000, .04, 300 );
            "444-44-4444", 16.75, 40 );

        String output = "";
// generically process each element in array employees
for ( int i = 0; i < employees.length; i++ ) {
    output += employees[ i ].toString();
    // determine whether element is a BasePlusCommissionEmployee
    if ( employees[ i ] instanceof BasePlusCommissionEmployee ) {
        // downcast Employee reference to BasePlusCommissionEmployee reference
        BasePlusCommissionEmployee currentEmployee = 
            ( BasePlusCommissionEmployee ) employees[ i ];
        double oldBaseSalary = currentEmployee.getBaseSalary();
        output += "old base salary: $" + oldBaseSalary;
        currentEmployee.setBaseSalary( 1.10 * oldBaseSalary );
        output += "new base salary with 10% increase is: $" +
            currentEmployee.getBaseSalary();
    } // end if
    output += "earned $" + employees[ i ].earnings() + "\n";
} // end for

// get type name of each object in employees array
for ( int j = 0; j < employees.length; j++ ) {
    output += "Employee " + j + " is a " +
        employees[ j ].getClass().getName();
    JOptionPane.showMessageDialog( null, output ); // display output
    System.exit( 0 );
} // end main

} // end class PayrollSystemTest
Downcast Operation

- Type of object must have an is-a relationship with type specified in the cast operator
  - otherwise ClassCastException occurs
- Object can be cast only to its own type or to the type of one of its superclasses.

Interfaces (1)

- are used to decouple method interfaces from their implementation.
- Interfaces are used for functionality not dependent on the class
- consist only of (implicit) abstract methods (= without implementation) and constants (implicit final static).
- no fields and no constructors

```java
public interface Sammlerstueck {
    public double sammlerWert();
    ...
}
```
Interfaces (2)

- A class may implement one or several interfaces.
- If a class implements an interface, then all of the interface’s methods must be implemented otherwise the class must be declared abstract.

```java
public class BriefMarke implements Sammlerstueck {
    ...
    public double sammlerWert() { //Implementierung
        ...
    }
}
```

```java
public class OldTimer extends Auto implements Sammlerstueck {
    ...
    public double sammlerWert() { //Implementierung
        ...
    }
}
```

Interfaces and Inheritance (1)

an example

```
interface X {
    void setup();
}
interface Y extends X {
    void status();
}
interface Z extends X {
    void check();
}
class C implements Y,Z {
    public void setup() { ... } // implementation
    public void status() { ... } // implementation
    public void check() { ... } // implementation
}
```
Interfaces and Inheritance (2)

- can be extended similar as classes.
- in contrast to classes, an interface may extend one or several other interfaces.

```java
interface A {
    void a();
}
interface B {
    void b();
}
interface C extends A, B { // C inherits a() and b()
}
```

Interfaces: Assignment Compatibility

- An object of a class A that implements an interface, can be assigned to a variable of the type of that interface.
- An interface cannot be instantiated.

```java
BriefMarke blaueMauritius = new Briefmarke();
OldTimer chevy57 = new OldTimer();
Sammlerstueck[] sammlung;
...
sammlung[i] = blaue Mauritius;
...
Sammlung[j] = chevy57;
...
for (i=0; i<sammlung.length; i++)
    gesamtWert += sammlung[i].sammlerWert();
```
Interfaces and Inheritance

- The implementation of an interface is an attribute which is inherited by subclasses.

- If a class $A$ implements an interface $I$, then the clause `implements $I$` at a subclass of $A$ is optional.

- Based on a class declaration one cannot always determine, whether a specific interface is implemented by that class.

Interfaces: Advantages

- Interface of a class is separated from implementation -> Software development can be handled more flexible.

- A variable of a type of a certain interface can be assigned objects of classes that implement this interface.
  - Different implementation variations can be dynamically selected or modified.
Interfaces: Importance

- Interfaces play an important role in the Java API, e.g.:
  - Event handling: EventListener, ActionListener, KeyListener, MouseListener, MouseMotionListener, ...
  - Collections: Collection, Iterator, Enumeration, ...
  - Objektserialisierung: Serializable
  - Multithreading: Runnable

Case Study: Creating and Using Interfaces

Shape.java

```java
// Fig. 10.18: Shape.java
// Shape interface declaration.

public interface Shape {
    public double getArea();    // calculate area
    public double getVolume();  // calculate volume
    public String getName();    // return shape name
}
// end interface Shape
```

Classes that implement Shape must implement these methods
// Fig. 10.19: Point.java
// Point class declaration implements interface Shape.

public class Point extends Object implements Shape {
    private int x;  // x part of coordinate pair
    private int y;  // y part of coordinate pair

    // no-argument constructor; x and y default to 0
    public Point()
    {
        // implicit call to Object constructor occurs here
    }

    // constructor
    public Point( int xValue, int yValue )
    {
        // implicit call to Object constructor occurs here
        x = xValue;  // no need for validation
        y = yValue;  // no need for validation
    }

    // set x in coordinate pair
    public void setX( int xValue )
    {
        x = xValue;  // no need for validation
    }

    // return x from coordinate pair
    public int getX()
    {
        return x;
    }

    // set y in coordinate pair
    public void setY( int yValue )
    {
        y = yValue;  // no need for validation
    }

    // return y from coordinate pair
    public int getY()
    {
        return y;
    }
}
// declare abstract method getArea
public double getArea() {
    return 0.0;
}

// declare abstract method getVolume
public double getVolume() {
    return 0.0;
}

// override abstract method getName to return "Point"
public String getName() {
    return "Point";
}

// override toString to return String representation of Point
public String toString() {
    return "[" + getX() + ", " + getY() + "]";
}

} // end class Point
// aim arrayOfShapes[ 0 ] at subclass Point object
arrayOfShapes[ 0 ] = point;

// aim arrayOfShapes[ 1 ] at subclass circle object
arrayOfShapes[ 1 ] = circle;

// aim arrayOfShapes[ 2 ] at subclass Cylinder object

// loop through arrayOfShapes to get name, string representation, area and volume of every shape in array
for ( int i = 0; i < arrayOfShapes.length; i++ ) {
    output += "\n\n" + arrayOfShapes[ i ].getName() + ":\nArea = " +
    twoDigits.format( arrayOfShapes[ i ].getArea() ) +
    "\nVolume = " +
    twoDigits.format( arrayOfShapes[ i ].getVolume() );
}

JOptionPane.showMessageDialog( null, output );  // display output
System.exit( 0 );

} // end main

} // end class InterfaceTest

// loop through arrayOfShapes to get name, string representation, area and volume of every shape in array

Case Study: Creating and Using Interfaces (Cont.)

• Implementing Multiple Interface
  – classes can implement as many interfaces as needed
  – Provide common-separated list of interface names after keyword implements

• Declaring Constants with Interfaces
  – public interface Constants {
    public static final int ONE = 1;
    public static final int TWO = 2;
    public static final int THREE = 3;
  }
  – classes that implement interface Constants can use ONE, TWO, and THREE anywhere in the class declaration.

Nested and Inner Classes

• Top-level classes
  – Not declared inside a class or a method
• Nested classes
  – Declared inside other classes
  – Nested classes can be static.
  – Inner classes are non-static nested classes
• Inner classes are frequently used for event handling or hiding implementation
• Inner classes:
  – can directly access its outer class’s members
  – reference this inside an inner class refers to current inner-class object.
  – refer to outer-class this by OuterClassName.this
• Inner classes can be declared private, protected, public or package access (default)
• Outer class is responsible for creating inner class objects
• Nested classes can be declared static
public class Time {
    private int hour; // 0 - 23
    private int minute; // 0 - 59
    private int second; // 0 - 59

    // one formatting object to share in toString and toUniversalString
    private static DecimalFormat twoDigits = new DecimalFormat( "00" );

    // Time constructor initializes each instance variable to zero;
    // ensures that Time object starts in a consistent state
    public Time() {
        this( 0, 0, 0 ); // invoke Time constructor with three arguments
    }

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

    // Time constructor: hour, minute and second supplied
    public Time( int h, int m, int s ) {
        setTime( h, m, s ); // invoke Time constructor with three arguments
    }

    // Time constructor: another Time3 object supplied
    public Time( Time time ) {
        this( time.getHour(), time.getMinute(), time.getSecond() );
    }

    // Set Methods
    // set a new time value using universal time; perform
    // validity checks on data; set invalid values to zero
    public void setTime( int h, int m, int s ) {
        setHour( h ); // set the hour
        setMinute( m ); // set the minute
        setSecond( s ); // set the second
    }

    // Get Methods
    public int getHour() { return hour; }
    public int getMinute() { return minute; }
    public int getSecond() { return second; }

    // Formatting
    public String toUniversalString() {
        return twoDigits.format( hour ) + ":" +
            twoDigits.format( minute ) + ":" +
            twoDigits.format( second );
    }

    public String toString() {
        return toUniversalString();
    }
}

// Fig. 10.21: Time.java
// Time class declaration with set and get methods.
import java.text.DecimalFormat;

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

    // one formatting object to share in toString and toUniversalString
    private static DecimalFormat twoDigits = new DecimalFormat( "00" );

    // Time constructor initializes each instance variable to zero;
    // ensures that Time object starts in a consistent state
    public Time() {
        this( 0, 0, 0 ); // invoke Time constructor with three arguments
    }

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

    // Time constructor: hour, minute and second supplied
    public Time( int h, int m, int s ) {
        setTime( h, m, s ); // invoke Time constructor with three arguments
    }

    // Time constructor: another Time3 object supplied
    public Time( Time time ) {
        this( time.getHour(), time.getMinute(), time.getSecond() );
    }

    // Set Methods
    // set a new time value using universal time; perform
    // validity checks on data; set invalid values to zero
    public void setTime( int h, int m, int s ) {
        setHour( h ); // set the hour
        setMinute( m ); // set the minute
        setSecond( s ); // set the second
    }

    // Get Methods
    public int getHour() { return hour; }
    public int getMinute() { return minute; }
    public int getSecond() { return second; }

    // Formatting
    public String toUniversalString() {
        return twoDigits.format( hour ) + ":" +
            twoDigits.format( minute ) + ":" +
            twoDigits.format( second );
    }

    public String toString() {
        return toUniversalString();
    }
}
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79

80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109

// validate and set hour
public void setHour( int h )
{
hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
}

Time.java

// validate and set minute
public void setMinute( int m )
{
minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
}
// validate and set second
public void setSecond( int s )
{
second = ( ( s >= 0 && s < 60 ) ? s : 0 );
}
// Get Methods
// get hour value
public int getHour()
{
return hour;
}

// get minute value
public int getMinute()
{
return minute;
}
// get second value
public int getSecond()
{
return second;
}
// convert to String in universal-time format
Override method
public String toUniversalString()
{
java.lang.Object.toString
return twoDigits.format( getHour() ) + ":" +
twoDigits.format( getMinute() ) + ":" +
twoDigits.format( getSecond() );
}
// convert to String in standard-time format
public String toString()
{
return ( ( getHour() == 12 || getHour() == 0 ) ?
12 : getHour() % 12 ) + ":" + twoDigits.format( getMinute() ) +
":" + twoDigits.format( getSecond() ) +
( getHour() < 12 ? " AM" : " PM" );
}
} // end class Time

Time.java
Lines 101-107
Override method
java.lang.Objec
t.toString


**TimeTestWindow.java**

**Line 7**  
*JFrame* provides basic window attributes and behaviors

**Line 17**  
*JFrame* (unlike *JApplet*) has constructor

**Line 19**  
Instantiate *Time* object

**Line 20**  
Create an instance of inner-class that implements *ActionListener*.

**Line 38**  
Instantiate object of inner-class that implements *ActionListener*.

```java
// Fig. 10.22: TimeTestWindow.java
// Inner class declarations used to create event handlers.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class TimeTestWindow extends JFrame {
    private Time time;
    private JLabel hourLabel, minuteLabel, secondLabel;
    private JTextField hourField, minuteField, secondField, displayField;
    private JButton exitButton;

    // set up GUI
    public TimeTestWindow() {
        // call JFrame constructor to set title bar string
        super( "Inner Class Demonstration" );

        // use inherited method getContentPane to get window's content pane
        Container container = getContentPane();
        container.setLayout( new FlowLayout() );  // change layout

        // set up hourLabel and hourField
        hourLabel = new JLabel( "Set Hour" );
        hourField = new JTextField( 10 );
        container.add( hourLabel );
        container.add( hourField );

        // set up minuteLabel and minuteField
        minuteLabel = new JLabel( "Set Minute" );
        minuteField = new JTextField( 10 );
        container.add( minuteLabel );
        container.add( minuteField );

        // set up secondLabel and secondField
        secondLabel = new JLabel( "Set Second" );
        secondField = new JTextField( 10 );
        container.add( secondLabel );
        container.add( secondField );

        // set up displayField
        displayField = new JTextField( 30 );
        displayField.setEditable( false );
        container.add( displayField );

        // set up exitButton
        exitButton = new JButton( "Exit" );
        container.add( exitButton );

        // create an instance of inner-class that implements ActionListener
        ActionEventHandler handler = new ActionEventHandler();
    }
}
```
Register ActionEventHandler with GUI components.

```java
// register event handlers; the object referenced by handler
// is the ActionListener, which contains method actionPerformed
// that will be called to handle action events generated by
// hourField, minuteField, secondField and exitButton
hourField.addActionListener( handler );
minuteField.addActionListener( handler );
secondField.addActionListener( handler );
exitButton.addActionListener( handler );
// end constructor
// display time in displayField
public void displayTime() {
    displayField.setText( "The time is: " + time );
}
// launch application: create, size and display TimeTestWindow;
// when main terminates, program continues execution because a
// window is displayed by the statements in main
public static void main( String args[] ) {
    TimeTestWindow window = new TimeTestWindow();
    window.setSize( 400, 140 );
    window.setVisible( true );
} // end main
```

---

Line 85
Declare inner class that implements ActionListener interface
Must implement method actionPerformed of ActionListener
When user presses JButton or Enter key, method actionPerformed is invoked
Determine action depending on where event originated

Lines 91-113
Determine action depending on where event originated
TimeTestWindow.java

```java
108     else if ( event.getSource() == secondField ) {
109         time.setSecond( Integer.parseInt( event.getActionCommand() ) );
110         secondField.setText( "" );
111     }
112
113     displayTime(); // call outer class's method
114 }
115
116     } // end inner class ActionEventHandler
117 }
118 } // end class TimeTestWindow
```

TimeTestWindow.java
Type-Wrapper Classes for Primitive Types

• Type-wrapper class
  – Each primitive type has one
    • Character, Byte, Intege, Boolean, Short, Long, Float, and Double.
  – Enable to represent primitive as Object
    • Primitive types can be processed polymorphically
  – Every type wrapper class is declared as final and extends class Number.
  – Methods of final classes are implicitly final (cannot be overwritten).

Summary

• Inheriting from concrete classes (extends) means inheriting both implementation and type.
• Polymorphism allows to create programs to process objects of types that might not exist when the program is under development.
• Abstract classes can be declared for which no objects ever will be created.
• Interfaces allow a strict separation of implementation from a type.
• Abstract classes are a mixture of inheritance and interfaces which allow to inherit a type and partially also implementation.
• Inner classes is another mechanism for hiding implementation.
Part 13
Exception Handling

Main concepts to be covered

• Defensive programming.
  – Anticipating that things could go wrong.
• Exception handling and throwing.
• Error reporting.
Motivation (1)

- original simple control flow without considering potential errors

```latex
\begin{verbatim}
doA();
doB();
doC();
\end{verbatim}
```

Motivation (2)

- Extended source code with error handling disturbs original control flow.
- Error cases dominate program logic.
- Complexity may prevent programmers from dealing with error cases.
- Note that reading EOF, last element of list (-1), etc. are not errors but normal control flow.

```latex
\begin{verbatim}
doA();
if (doA erroneous)
    treat error cases for doA;
else
    doB();
    if (doB erroneous)
        treat error cases for doB;
    else
        doC();
        if (doC erroneous)
            treat error cases for doC;
        else etc.
\end{verbatim}
```
Exception-Handling Overview

• Uses of exception handling
  – Process exceptions from program components
  – Handle exceptions in a uniform manner in large projects
  – Remove error-handling code from “main line” of execution
• A method detects an error and throws an exception
  – Exception handler processes the error
  – Uncaught exceptions yield adverse effects
    • Might terminate program execution

Exception-Handling Overview

• Code that could generate errors put in try blocks
  – Code for error handling enclosed in a catch clause
  – The finally clause always executes
• Termination model of exception handling
  – The block in which the exception occurs terminates before dealing with the exception
• throws clause specifies exceptions that a method throws
Some causes of error situations

- Incorrect implementation.
  - Does not meet the specification.
- Inappropriate object request.
  - E.g., invalid index.
- Inconsistent or inappropriate object state.
  - E.g. arising through class extension.

Not always programmer error

- Errors often arise from the environment:
  - Incorrect URL entered.
  - Network interruption.
- File processing is particular error-prone:
  - Missing files.
  - Lack of appropriate permissions.
Exploring errors

• Explore error situations through the *address-book* projects.
• Two aspects:
  – Error reporting.
  – Error handling.

Defensive programming

• Client-server interaction.
  – Should a server assume that clients are well-behaved?
  – Or should it assume that clients are potentially erroneous?
• Significant differences in implementation required.
**Issues to be addressed**

- How much checking by a server on method calls?
- How to report errors?
- How can a client anticipate failure?
- How should a client deal with failure?

**An Example**

- Create an AddressBook object.
- Try to remove a non-existing entry in empty AddressBook.
- A runtime error results.
  - Whose ‘fault’ is this?
- Anticipation and prevention are preferable to apportioning blame.
Argument values

- Arguments represent a major ‘vulnerability’ for a server object.
  - Constructor arguments initialize state.
  - Method arguments often contribute to behavior.
- Argument checking is one defensive measure.

Server Error Reporting

- How to report illegal arguments?
  - To the user?
    - Is there a human user?
    - Can they solve the problem?
  - To the client object?
    - Return a diagnostic value.
    - *Throw an exception*
Client responses

- Test the return value.
  - Attempt recovery on error.
  - Avoid program failure.
- Ignore the return value.
  - Cannot be prevented.
  - Likely to lead to program failure.
- Exceptions are preferable.

Exception-throwing principles

- A special language feature.
- No ‘special’ return value needed.
- Errors cannot be ignored in the client.
  - The normal flow-of-control is interrupted.
- Specific recovery actions are encouraged.
/**
 * Look up a name or phone number and return the corresponding contact details.
 * @param key The name or number to be looked up.
 * @return The details corresponding to the key, or null if there are none matching.
 * @throws NullPointerException if the key is null.
 */
public ContactDetails getDetails(String key)
{
    if(key == null){
        throw new NullPointerException("null key in getDetails");
    }
    return (ContactDetails) book.get(key);
}

• An exception object is constructed:
  - new ExceptionType("...");
• The exception object is thrown:
  - throw ...
Inheritance hierarchy for class Throwable

The exception class hierarchy
Exception categories

- Checked exceptions
  - Subclass of Exception
  - Use for anticipated failures.
  - Where recovery may be possible.
  - compiler statically checks methods whether checked exceptions are thrown
    - “Exception must be caught or declared in throws clause”
  - no compilation if exceptions are ignored.

- Unchecked exceptions
  - Subclass of RuntimeException
  - exceptions typically cannot be handled by a program but can be prevented through proper coding (check for denominator != 0 to avoid ArithmeticException)
  - use for unanticipated failures
  - where recovery is unlikely
  - no static compiler analysis
  - use throw construct for throwing unchecked exceptions
  - exceptions will not be caught – program terminates

The effect of an exception

- The throwing method finishes prematurely.
- No return value is returned.
- Control does not return to the client’s point of call.
  - So the client cannot carry on regardless.
- A client may ‘catch’ an exception.
Unchecked exceptions

- Use of these is ‘unchecked’ by the compiler because they require debugging and program modification.
- Cause program termination if not caught.
  - This is the normal practice.
- `IllegalArgumentException`, `ArithmeticException`, `IndexOutOfBoundsException` are typical examples.
- no need to specify such exceptions in a method’s `throws` clause.
- optional to use try-catch
- exceptions are passed on to calling methods until proper catch is found
- if no catch is found then Java runtime system reports exception – program will be terminated.
- Applets or programs using AWT continue to execute – results cannot be trusted anymore.

---

Argument checking

```java
public ContactDetails getDetails(String key)
{
    if(key == null) {
        throw new NullPointerException("null key in getDetails");
    }
    if(key.trim().length() == 0) {
        throw new IllegalArgumentException("Empty key passed to getDetails");
    }
    return (ContactDetails) book.get(key);
}
```
Preventing object creation

```java
public ContactDetails(String name, String phone, String address) {
    if(name == null) {
        name = "";
    }
    if(phone == null) {
        phone = "";
    }
    if(address == null) {
        address = "";
    }

    this.name = name.trim();
    this.phone = phone.trim();
    this.address = address.trim();

    if(this.name.length() == 0 && this.phone.length() == 0) {
        throw new IllegalStateException(  
            "Either the name or phone must not be blank.";
        )
    }
}
```

Exception Handling for Checked Exceptions

- Checked exceptions are meant to be caught.
- The compiler ensures that their use is tightly controlled.
  - In both server and client.
- Used properly, failures may be recoverable.
- 2 options:
  - direct: try-catch
  - indirect: throw exception
    - Calling method must deal with exception (try-catch) or pass it on to the next method in calling hierarchy.
Exception Handling for Checked Exceptions

- 2 options to deal with checked exceptions:
  - direct: try-catch
  ```java
  try {
    inputFile = new BufferedReader(new FileReader("file.txt"));
  } catch (FileNotFoundException e) {
    errorField.setText("File missing - " + "enter file name again");
  }
  ```
  
  - indirect: throws exception
    - Calling method must deal with exception (try-catch) or pass it on to the next method in calling hierarchy.
  ```java
  private void openFile() throws IOException {
    inputFile = new BufferedReader(new FileReader("file.txt"));
  }
  ```

The throws clause

- Methods throwing a checked exception must include a throws clause:
  ```java
  public void saveToFile(String destinationFile) throws IOException {
  ```

- These exceptions may be handled by calling methods.

- A method M1 that calls other methods which throw checked exceptions, then those exceptions must be caught or declared (by throws clause) in M1.
The try block

• Clients catching an exception must protect the call with a try block:

```java
try {
    Protect one or more statements here.
} catch(Exception e) {
    Report and recover from the exception here.
    throw exception; // rethrow for further processing
}
```

• Rethrow exception if `catch` cannot handle it

```java
try{
    addressbook.saveToFile(filename);
    tryAgain = false;
}
catch(IOException e) {
    System.out.println("Unable to save to " + filename);
    tryAgain = true;
}
```
Catching multiple exceptions
Java compiler enforces to place more general exceptions at the end of catch block.

```java
try {
    ...
    ref.process();
    ...
}
catch(EOFException e) {
    // Take action on an end-of-file exception.
    ...
}
catch(FileNotFoundException e) {
    // Take action on a file-not-found exception.
    ...
}
catch(Exception e) {
    // treat all other exceptions
}
```

The finally clause

- optional clause placed after last catch clause
- A finally clause is executed even if a return statement is executed in the try or catch clauses.
- A uncaught or propagated exception still exits via the finally clause.
- Commonly resource release code is placed in finally clauses to avoid resource leaks.
  - files, database and network connections
The finally clause

try {
    Protect one or more statements here.
}
catch(Exception e) {
    Report and recover from the exception here.
}
finally {
    Perform any actions here common to whether or not an exception is thrown.
}
UsingExceptions.java

```java
// catch exception thrown in try block
    catch ( Exception exception ) {
        System.err.println("Exception handled in method throwException");
        throw exception; // rethrow for further processing
    }
    // any code here would not be reached

// this block executes regardless of what occurs in try/catch
finally {
    System.err.println("Finally executed in throwException");
}
// any code here would not be reached - without throw statement
    // in line 32, this code would be reached.
} // end method throwException

// demonstrate finally when no exception occurs
public static void doesNotThrowException() {
    // try block does not throw an exception
    try {
        System.out.println("Method doesNotThrowException");
    }
    // catch does not execute, because no exception thrown
    catch ( Exception exception ) {
        System.err.println( exception );
    }
    // this clause executes regardless of what occurs in try/catch
    finally {
        System.err.println("Finally executed in doesNotThrowException");
    }
    System.out.println("End of method doesNotThrowException");
} // end method doesNotThrowException
// end class UsingExceptions
```
Defining new exceptions

- **Extend** `Exception` or `RuntimeException`.
- Define new types to give better diagnostic information.
  - Include reporting and/or recovery information.

```java
public class NoMatchingDetailsException extends Exception {
    private String key;

    public NoMatchingDetailsException(String key) {
        this.key = key;
    }

    public String getKey() {
        return key;
    }

    public String toString() {
        return "No details matching '" + key + "' were found.";
    }
}
```
Error recovery

• Clients should take note of error notifications.
  – Check return values.
  – Don’t ‘ignore’ exceptions.
• Include code to attempt recovery.
  – Will often require a loop.

Attempting recovery

// Try to save the address book.
boolean successful = false;
int attempts = 0;
do {
  try {
    addressbook.saveToFile(filename);
    successful = true;
  }
  catch(IOException e) {
    System.out.println("Unable to save to " + filename);
    attempts++;
    if(attempts < MAX_ATTEMPTS) {
      filename = an alternative file name;
    }
  }
} while(!successful && attempts < MAX_ATTEMPTS);
if(!successful) {
  Report the problem and give up;
}
Text input-output

• Input-output is particularly error-prone.
  – It involves interaction with the external environment.
• The java.io package supports input-output.
• java.io.IOException is a checked exception.

Readers, writers, streams

• Readers and writers deal with textual input.
  – Based around the char type.
• Streams deal with binary data.
  – Based around the byte type.
• The address-book-io project illustrates textual IO.
Text output

- Use the FileWriter class.
  - Open a file.
  - Write to the file.
  - Close the file.
- Failure at any point results in an IOException.

```java
try {
    FileWriter writer = new FileWriter("name of file");
    while (there is more text to write) {
        ...
        writer.write(next piece of text);
        ...
    }
    writer.close();
} catch (IOException e) {
    something went wrong with accessing the file
}
```
Text input

- Use the FileReader class.
- Augment with BufferedReader for line-based input.
  - Open a file.
  - Read from the file.
  - Close the file.
- Failure at any point results in an IOException.

```java
try {
    BufferedReader reader = new BufferedReader(
            new FileReader("name of file "));
    String line = reader.readLine();
    while(line != null) {
        do something with line
        line = reader.readLine();
    }
    reader.close();
} catch(FileNotFoundException e) {
    the specified file could not be found
} catch(IOException e) {
    something went wrong with reading or closing
```
Stack Unwinding

- Exception not caught in scope
  - Method terminates
  - Stack unwinding occurs
  - Another attempt to catch exception in the next outer try/catch statement.
  - Exception must be handled before normal program execution can continue.
  - If no catch handler ever catches an exception then program terminates.

```java
// Fig. 15.4: UsingExceptions.java
// Demonstration of stack unwinding.
public class UsingExceptions {
    public static void main( String args[] )
    {
        // call throwException to demonstrate stack unwinding
        try {
            throwException();
        }
        // catch exception thrown in throwException
        catch ( Exception exception ) {
            System.err.println( "Exception handled in main" );
        }
    }
    // throwException throws exception that is not caught in this method
    public static void throwException() throws Exception
    {
        // throw an exception and catch it in main
        try {
            System.out.println( "Method throwException" );
            throw new Exception();
        }
    }
}
```
UsingExceptions.java

```java
// catch is incorrect type, so Exception is not caught
catch ( RuntimeException runtimeException ) {
    System.err.println("Exception handled in method throwException");
}

// finally clause always executes
finally {
    System.err.println("Finally is always executed");
}
```
public class UsingExceptions {
    public static void main(String args[])
    {
        try {
            method1(); // call method1
        }
        catch (Exception exception) {
            System.err.println( exception.getMessage() + "\n" );
            exception.printStackTrace();
            StackTraceElement[] traceElements = exception.getStackTrace();
            System.out.println( "\nStack trace from getStackTrace:" );
            System.out.println( "Class		File			Line	Method" );
            for (int i = 0; i < traceElements.length; i++) {
                StackTraceElement currentElement = traceElements[i];
                System.out.print( currentElement.getClassName() + "\t" );
                System.out.print( currentElement.getFileName() + "\t" );
                System.out.print( currentElement.getLineNumber() + "\t" );
                System.out.print( currentElement.getMethodName() + "\n" );
            }
        }
        catch method2; throw exceptions back to main
        public static void method1() throws Exception
        {
            method2();
        }
        // call method2; throw exceptions back to method1
        public static void method2() throws Exception
        {
            method3();
        }
        // throw Exception back to method2
        public static void method3() throws Exception
        {
            throw new Exception( "Exception thrown in method3" );
        }
    }
}
Constructors and Exception Handling

- What if error happens in constructor?
  - E.g. unable to allocate memory for storing object
  - Object not in a consistent state.

- Constructor cannot return a value

- Throw exception if constructor causes error
  - Indicate problem that occurred
  - Handle problem properly
Summary (1)

- Runtime errors arise for many reasons.
  - An inappropriate client call to a server object.
  - A server unable to fulfill a request.
  - Programming error in client and/or server.
- Runtime errors often lead to program failure.
- Defensive programming anticipates errors – in both client and server.
- Exceptions provide a reporting and recovery mechanism.

Summary (2)

- Exception handling to catch and handle errors
  - Fault tolerant programming
- Examples: out-of-bounds array index, arithmetic overflow, division by zero, not enough memory, etc.
- Java’s exception handling mechanism:
  - Remove error processing from main program code
- Do not place try/catch/finally around every statement
- Place one try block around significant code portion
- Following try block place all catch clauses
- Release resources in finally clause
- If handler catches exception which he cannot deal with, then re-throw it
Part 14
Multithreading

Prozesse

– Was ist ein Prozess?
  • ein exekutierendes Programm (aktive Einheit)
  • ein Prozess benötigt Ressourcen:
    – CPU-Zeiten, Speicher, Files, I/O Systeme
  • Betriebssystem ermöglicht:
    – Erzeugen und Terminieren von Prozessen
    – Scheduling von Prozessen
    – Synchronisierung von Prozessen
    – Kommunikation zwischen Prozessen
    – Behandlung von Deadlocks

– Was ist kein Prozess?
  • Programm (passive Einheit) auf einer Speicherplatte
– Prozess kann mehrere Threads erzeugen.
Threads

- Thread: Leichtgewichtprozess, der aus folgenden Komponenten besteht:
  - Programmzähler
  - Register-Set für thread-lokale Variablen
  - Stack
  - Zustand

- Thread
  - ist immer Teil eines Prozesses
  - hat Zugriff auf Speicher und Ressourcen seines Prozesses.
  - befindet sich in einem Ausführungs Zustand (Ready, etc.).
  - kann Prioritäten haben.

- Eine Menge von Threads teilt sich
  - Speicher (außer Stack)
  - Programmcode
  - offene Files (inkl. Daten in File-Buffer)
  - Signale

Thread-Verwendung
Textverarbeitungsprogramm mit 3 Threads
Thread-Verwendung
Ein multithreaded Web Server

Threads: Vor- u. Nachteile

• Vorteile:
  – sehr schnelles Context Switching
  – Datenaustausch mithilfe eines gemeinsamen Speichers (keine Kernelaufrufe für expliziten Datenaustausch)
  – Wenn ein Thread blockiert, können andere Threads weiterarbeiten.
  – asynchrone Kommunikation mittels Threads (z.B. Java)
  – einfache Programmierung von nebenläufigen Aktivitäten

• Nachteile:
  – Komplexität von nebenläufigen Aktivitäten (Synchronisierung, Deadlocks, etc.)
Multithreaded Prozessmodell

Thread-Modell (1)

(a) 3 Prozesse jeder mit einem Thread
(b) 1 Prozess mit 3 Threads
Thread-Modell (2)

• Jeder Thread mit seinem eigenen Stack

Benutzer-Threads

• Der Kernel sieht nur den Prozess aber keine Threads.
• Verwaltung der Threads mittels Thread Library, die außerhalb des Betriebssystems liegt.
• Context-Switching zwischen Threads im User Mode ohne Betriebssystem.
• Vorteile:
  – Applikationsspezifisches Thread Scheduling ist möglich.
  – Benutzer-Threads sind schneller als Kernel-Threads, weil der Kernel nicht involviert ist.
• Nachteile:
  – Kein Parallelismus auf Multiprozessorarchitekturen
  – Wenn ein Thread einen System Call (immer blockierend) exekutiert, dann wird der ganze Prozess blockiert.
Implementierung eines Threads im Benutzeradressraum

Kernel Threads

- Thread Management des Betriebssystems verwaltet Threads.
- API zum Kernel
- Alle Threads eines Prozesses sind für den Kernel sichtbar.
- Thread Scheduling durch Kernel.
- Vorteile:
  - Wenn Thread blockiert, können andere Threads weiterarbeiten.
  - Parallelismus auf Multiprozessorarchitekturen kann ausgenutzt werden.
- Nachteile:
  - Thread Context Switching erfordert Umschalten zwischen 2 Modi
  - Erhöht die Zeit für das Context Switching.
    - User Mode <-> Kernel Mode

- Ideal wäre Kombination von User mit Kernel Threads (z.B. Sun Solaris).
Multithreading in Java

- Java includes multithreading primitives as part of the language for manipulating threads in a portable manner across platforms
  - package java.lang
- Java provides built-in multithreading
  - Programmer specifies a program that contains multiple threads of execution.
  - Each thread designates a portion of a program which may execute in parallel with other threads.
  - Multithreading improves the performance of some programs
    - multiprocessor versus single processor architecture
- C and C++ do not have built-in multithreading
  - employ thread libraries (not always portable)
Thread States: Life Cycle of a Thread

- Thread states
  - Spawn state
    - Thread was just created
  - Ready state
    - Thread’s `start` method invoked
    - Thread can now execute
  - Running state
    - Thread is assigned a processor and running
  - Blocked
    - wait for I/O operation to finish; wait for lock of an object;
  - Waiting
    - wait for event to occur
  - Sleeping
    - sleep for some time interval
  - Finish state
    - Thread has completed or exited
    - Eventually disposed of by system

Thread life-cycle statechart diagram
Thread Priorities and Thread Scheduling

- **Java thread priority**
  - Priority in range 1 (minimum) – 10 (maximum)
  - Default priority Thread.Norm_PRIORITY (5)
  - Thread priorities do not guarantee order of execution.
  - New threads inherit priority of generating thread.

- **Timeslicing (preemptive scheduling)**
  - Each thread assigned constant time slice on the processor (called a quantum).
  - Keeps highest priority threads running.
  - Threads with equal priority are run in round robin fashion
  - Starvation is possible if new threads with higher priorities are being generated.

Thread priority scheduling example
Multithreading

• In Java threads are objects of class Thread or a subclass of this class.
• Classes whose objects should get a separate thread
  – derived from class Thread, or
  – implement interface runnable
• Method run must be implemented by a derived class of class Thread or a class that implements interface runnable.

Methods for Manipulating the State of Threads

• start()
  start thread by placing thread in ready state
• run()
  processor is assigned to thread (running state)
• sleep(long ms)
  sleep for ms milliseconds and then awaken
• notify()
  inform a waiting thread to transfer from wait to ready state
• wait()
  thread waits in the waiting state for some event to occur
• etc.
Interaction Among Threads

- Interaction among threads through accessing shared instance or class variables, or through invocation of methods visible in the run method.
- Synchronization:
  Maintain consistent shared data among set of threads.
- Java uses monitor concept and critical regions for synchronization:
  - declare methods as synchronized
  - synchronized statement

Creating and Executing Threads

```java
// Fig. 16.3: ThreadTester.java
// Multiple threads printing at different intervals.

public class ThreadTester {
    public static void main( String [] args ) {
        // create and name each thread
        PrintThread thread1 = new PrintThread( "thread1" );
        PrintThread thread2 = new PrintThread( "thread2" );
        PrintThread thread3 = new PrintThread( "thread3" );

        System.err.println( "Starting threads" );

        thread1.start(); // start thread1 and place it in ready state
        thread2.start(); // start thread2 and place it in ready state
        thread3.start(); // start thread3 and place it in ready state

        System.err.println( "Threads started, main ends\n" );
    }

} // end class ThreadTester
```

Outline

ThreadTester.java

Lines 9-11

Lines 15-17

create four PrintThreads

create four PrintThreads

call start methods

call start methods
25 // class PrintThread controls thread execution
26 class PrintThread extends Thread {
27     private int sleepTime;
28     // assign name to thread by calling superclass constructor
29     public PrintThread( String name ) {
30         super( name );
31     }
32     // pick random sleep time between 0 and 5 seconds
33     sleepTime = ( int ) ( Math.random() * 5001 );
34 }
35     // method run is the code to be executed by new thread
36     public void run() {
37         // put thread to sleep for sleepTime amount of time
38         try {
39             System.err.println( getName() + " going to sleep for " + sleepTime );
40             Thread.sleep( sleepTime );
41         }
42         // if thread interrupted during sleep, print stack trace
43         catch ( InterruptedException exception ) {
44             exception.printStackTrace();
45         }
46         // print thread name
47         System.err.println( getName() + " done sleeping" );
48     }
49     } // end class PrintThread
Thread Synchronization (1)

- If multiple threads read shared object -> no synchronization necessary
- If multiple threads read and at least one thread writes shared object -> synchronization necessary
- Synchronized access to shared objects is important to maintain consistent data
- Idea: exclusive access to shared data for each thread
- Java uses locks (monitors) for thread synchronization
Thread Synchronization (2)

- Every object has a lock (monitor) to control access to its methods and variables.
- The monitor allows one thread at a time to execute inside a synchronized method on the object.
- When program enters a synchronized method:
  - object is locked
- Only one synchronized method may be active on an object at any given time.
- All other threads trying to enter a synchronized method of an object are blocked.
- When synchronized method finished execution, the lock on the object is released:
  - Next highest priority thread may enter synchronized method.

Thread Synchronization (3)

- If a thread that obtained a lock on an object, determines that it is waiting for some event to continue execution:
  - invoke method wait()
- method wait()
  - Thread releases lock on the object and waits in Waiting state for the event to occur.
- method notify()
  - event occurs for which thread is waiting
  - notify() allows waiting thread to enter Ready state and try to acquire lock again.
  - notify() can only be executed by a non-waiting thread.
- method notifyAll()
  - all waiting threads become eligible to acquire lock again
- methods wait(), notify(), notifyAll() are inherited by all classes from class Object
Critical Region

Mutual exclusion through a critical region

Thread A

Thread B

Producer/Consumer Relationship without Synchronization

- Buffer
  - Shared memory region
- Producer thread
  - Generates data to add to buffer
- Consumer thread
  - Reads data from buffer
- First approach: no synchronization between producer and consumer.
1 // Fig. 16.4: Buffer.java
2 // Buffer interface specifies methods called by Producer and Consumer.
3
4 public interface Buffer {
5    public void set( int value );  // place value into Buffer
6    public int get();              // return value from Buffer
7 }

1 // Fig. 16.5: Producer.java
2 // Producer's run method controls a thread that
3 // stores values from 1 to 4 in sharedLocation.
4
5 public class Producer extends Thread {
6    private Buffer sharedLocation; // reference to shared object
7
8    // constructor
9    public Producer( Buffer shared )
10    {
11        super( "Producer" );
12        sharedLocation = shared;
13    }
14
15    // store values from 1 to 4 in sharedLocation
16    public void run()
17    {
18        for ( int count = 1; count <= 4; count++ ) {
19            // sleep 0 to 3 seconds, then place value in Buffer
20            try {
21                Thread.sleep( ( int ) ( Math.random() * 3001 ) );
22                sharedLocation.set( count );
23            }
24        }
25    }
Producer.java

```java
// if sleeping thread interrupted, print stack trace
    catch ( InterruptedException exception ) {
        exception.printStackTrace();
    }
} // end for
System.err.println( getName() + " done producing." + "\nTerminating " + getName() + ".");
} // end method run
} // end class Producer
```

Consumer.java

```java
// Fig. 16.6: Consumer.java
// Consumer's run method controls a thread that loops four
// times and reads a value from sharedLocation each time.

public class Consumer extends Thread {

    private Buffer sharedLocation; // reference to shared object

    // constructor
    public Consumer( Buffer shared ) {
        super( "Consumer" );
        sharedLocation = shared;
    }

    // read sharedLocation's value four times and sum the values
    public void run() {
        int sum = 0;
        for ( int count = 1; count <= 4; count++ ) {
            // sleep 0 to 3 seconds, read value from buffer and add to sum
            try {
                Thread.sleep( ( int ) ( Math.random() * 3001 ) );
                sum += sharedLocation.get();
            } catch ( InterruptedException exception ) {
                exception.printStackTrace();
            }
        }
    }
}
```
Consumer.java

```java
// if sleeping thread interrupted, print stack trace
catch (InterruptedException exception) {
    exception.printStackTrace();
}

System.err.println(getName() + " read values totaling: " + sum + ".\nTerminating " + getName() + ");
```
public class SharedBufferTest {
    public static void main(String[] args) {
        // create shared object used by threads
        Buffer sharedLocation = new UnsynchronizedBuffer();
        // create producer and consumer objects
        Producer producer = new Producer(sharedLocation);
        Consumer consumer = new Consumer(sharedLocation);
        producer.start(); // start producer thread
        consumer.start(); // start consumer thread
    }
}

Consumer reads -1
Producer writes 1
Consumer reads 1
Consumer reads 1
Consumer read values totaling: 2.
Terminating Consumer.
Producer writes 2
Producer writes 3
Producer writes 4
Producer done producing.
Terminating Producer.

Producer writes 1
Producer writes 2
Consumer reads 2
Producer writes 3
Consumer reads 3
Producer writes 4
Producer done producing.
Terminating Producer.
Consumer reads 4
Consumer reads 4
Consumer read values totaling: 13.
Terminating Consumer.
### Producer/Consumer Relationship with Synchronization

- Synchronize threads to ensure correct data
- Every data produced is first consumed before another data is placed into shared object.
- Producer thread
  - Generates data to add to buffer
  - Calls `wait` if consumer has not read previous message in buffer
  - Writes to empty buffer and calls `notify` for consumer
- Consumer thread
  - Reads data from buffer and calls `notify` for producer
  - Calls `wait` if buffer empty
This class implements the `Buffer` interface.

// SynchronizedBuffer synchronizes access to a single shared integer.

```java
public class SynchronizedBuffer implements Buffer {

  private int buffer = -1; // shared by producer and consumer threads
  private int occupiedBufferCount = 0; // count of occupied buffers

  // place value into buffer
  public synchronized void set(int value) {
    // for output purposes, get name of thread that called this method
    String name = Thread.currentThread().getName();

    // while there are no empty locations, place thread in waiting state
    while (occupiedBufferCount == 1) {
      // output thread information and buffer information, then wait
      try {
        System.err.println(name + " tries to write.");
        displayState("Buffer full. " + name + " waits.");
        wait();
      } catch (InterruptedException exception) {
        exception.printStackTrace();
      }
    }
    // if waiting thread interrupted, print stack trace
    } // end while
    buffer = value; // set new buffer value
    // indicate producer cannot store another value
    // until consumer retrieves current buffer value
    ++occupiedBufferCount;
    displayState(name + " writes " + buffer);
    notify(); // tell waiting thread to enter ready state
  }

  // return value from buffer
  public synchronized int get() {
    // for output purposes, get name of thread that called this method
    String name = Thread.currentThread().getName();
    return buffer;
  }
}
```
// while no data to read, place thread in waiting state
while (occupiedBufferCount == 0) {
    // output thread information and buffer information, then wait
    try {
        System.err.println(name + " tries to read.");
        displayState("Buffer empty. " + name + " waits.");
        wait();
    }
    catch (InterruptedException exception) {
        exception.printStackTrace();
    }
}

// indicate that producer can store another value
--occupiedBufferCount;

displayState(name + " reads " + buffer);
notify(); // tell waiting thread to become ready to execute
return buffer;

} // end method get; releases lock on SynchronizedBuffer

// display current operation and buffer state
public void displayState(String operation) {
    StringBuffer outputLine = new StringBuffer(operation);
    outputLine.setLength(40);
    outputLine.append(buffer + "\t\t" + occupiedBufferCount);
    System.err.println(outputLine);
    System.err.println();
}

} // end class SynchronizedBuffer
public class SharedBufferTest2 {
    public static void main(String[] args) {
        SynchronizedBuffer sharedLocation = new SynchronizedBuffer();
        // Display column heads for output
        StringBuffer columnHeads = new StringBuffer("Operation");
        columnHeads.setLength(40);
        columnHeads.append("Buffer\t\tOccupied Count");
        System.err.println(columnHeads);
        System.err.println();
        sharedLocation.displayState("Initial State");
        // create producer and consumer objects
        Producer producer = new Producer(sharedLocation);
        Consumer consumer = new Consumer(sharedLocation);
        producer.start(); // start producer thread
        consumer.start(); // start consumer thread
    }
} // end class SharedBufferTest2

<table>
<thead>
<tr>
<th>Operation</th>
<th>Buffer</th>
<th>Occupied Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial State</td>
<td>-1</td>
<td>0</td>
</tr>
</tbody>
</table>

Consumer tries to read. Buffer empty. Consumer waits. -1 0
Producer writes 1 1 1
Consumer reads 1 1 0
Consumer tries to read. Buffer empty. Consumer waits. 1 0
Producer writes 2 2 1
Consumer reads 2 2 0
Producer writes 3 3 1
Consumer reads 3                        3               0
Consumer tries to read.
Buffer empty. Consumer waits.           3               0
Producer writes 4                       4               1
Consumer reads 4                        4               0
Producer done producing.
Terminating Producer.
Consumer read values totaling: 10.
Terminating Consumer.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Buffer</th>
<th>Occupied Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial State</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>Consumer tries to read.</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>Buffer empty. Consumer waits.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Producer writes 1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Consumer reads 1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Producer writes 2</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Producer tries to write.
Buffer full. Producer waits.        2       1
Consumer reads 2                    2       0
Producer writes 3                    3       1
Consumer reads 3                    3       0
Producer writes 4                    4       1
Producer done producing.
Terminating Producer.
Consumer reads 4                    4       0
Consumer read values totaling: 10.
Terminating Consumer.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Buffer</th>
<th>Occupied Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial State</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>Producer writes 1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Consumer reads 1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Producer writes 2</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
### Synchronized Statement (1)

- Protect a block of statements from being executed by more than 1 thread.

- Entering the block of statements is protected by locking the object when program enters the synchronized statement.
public class ThreadSync extends Thread {
    static int cnt = 0;

    public static void main (String[] args) {
        Thread t1 = new ThreadSync();
        Thread t2 = new ThreadSync();
        t1.start();
        t2.start();
    }

    public void run() {
        while (true) {
            synchronized (getClass()) {
                increment();
                System.out.println(cnt);
            }
        }
    }
}

Notes:
At any point in time, only a single thread can reside in the critical region.

Method getClass() of class Object delivers an object of type Class which provides information about the object's type. This object exists exactly once for every class.

Runnable Interface (1)

• So far, we extended class Thread to create classes supporting multithreading.
• A class cannot extend more than one class.
• Implement Runnable Interface for multithreading support
• Code that executes the thread is placed in method run.
**Interface Runnable (2)**

1. **declare class that implements** interface Runnable
2. **instantiate this class**
3. **instantiate class Thread by passing an object of a class that implements** Runnable
4. **invoke method** `start()`

```java
class MyRunner implements Runnable {
    // 1
    public void run() { ... }
}

public class TestRunnable {
    public static void main () {
        MyRunner myRunner = new MyRunner();  // 2
        new Thread(myRunner).start();        // 3, 4
    }
}
```

**Summary**

- **Thread**: lightweight processes for fast context switching and simple programming of parallel systems.
- Each process may contain several threads.
- If a thread blocks, then other threads may continue execution.
- 3 types of thread implementations: user, kernel and hybrid threads.
- Synchronization among threads through synchronized methods and statements.