Presentation slides for

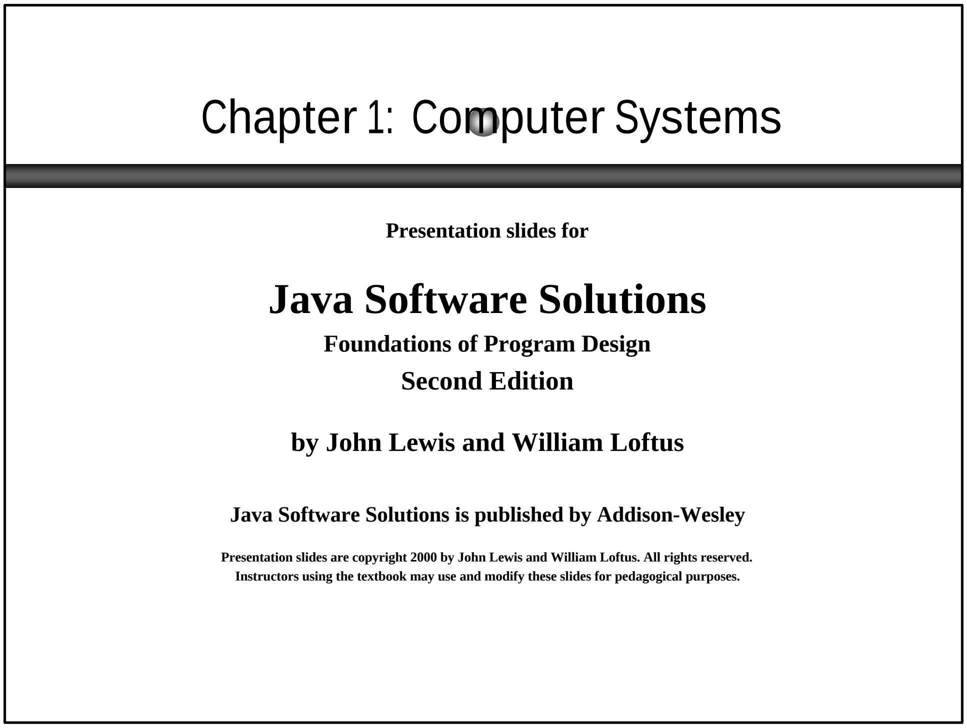
Java Software Solutions

Foundations of Program Design
Second Edition

by John Lewis and William Loftus

Part I
(chapters 1 – 5)

Prepared for Java Programming 90.301
by Marjan Trutschl
mtrutsch@cs.uml.edu



Chapter 1: Computer Systems

Presentation slides for

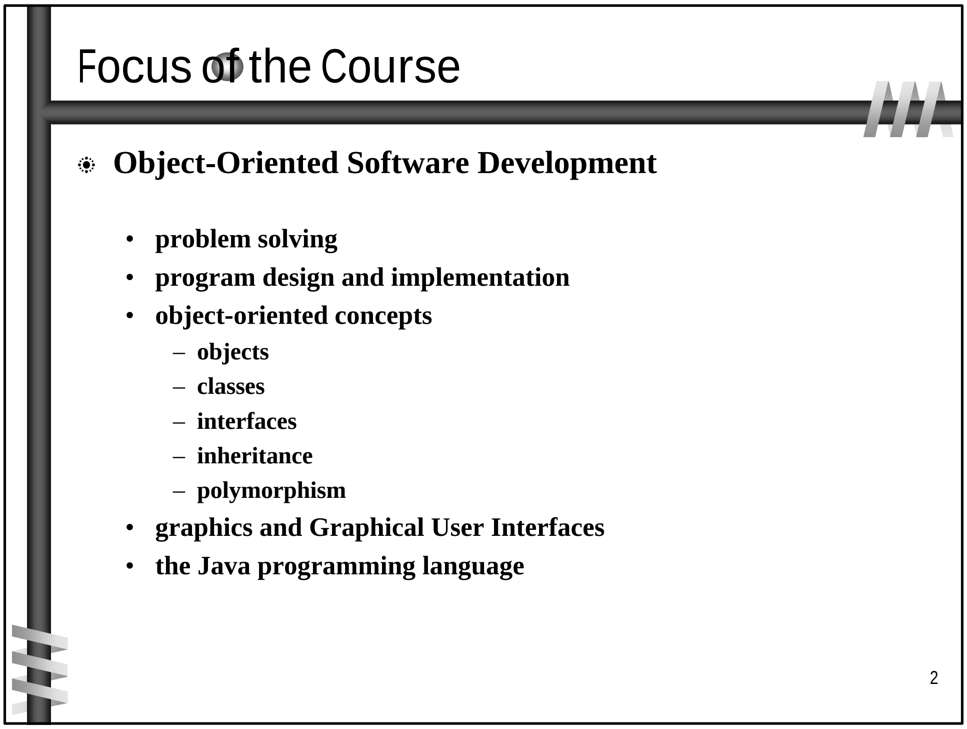
Java Software Solutions

Foundations of Program Design

Second Edition

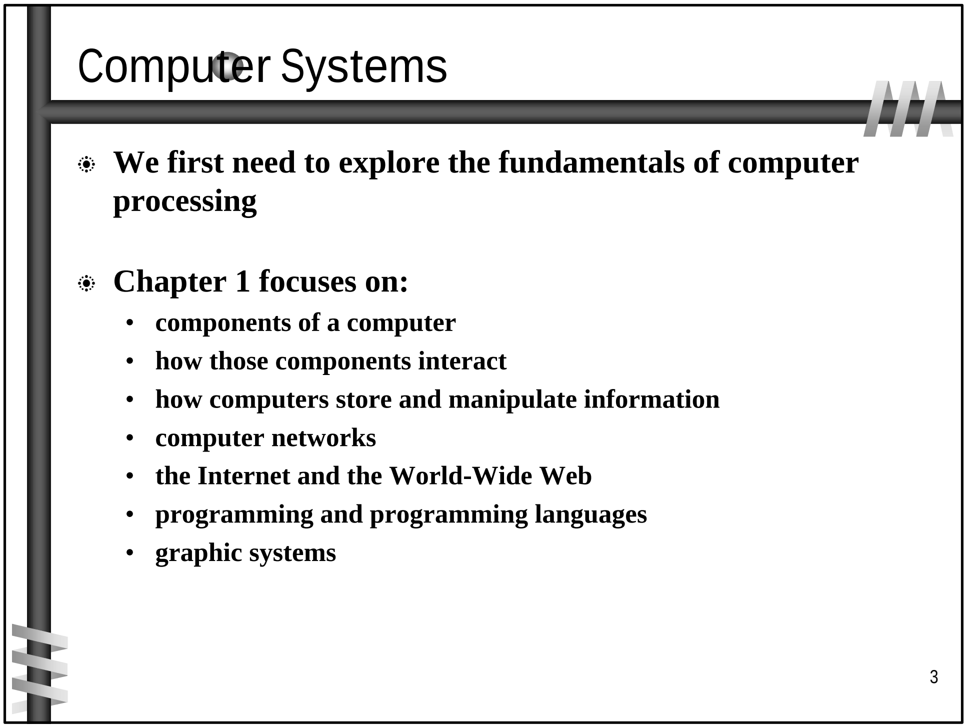
by John Lewis and William Loftus

Java Software Solutions is published by Addison-Wesley



Focus of the Course

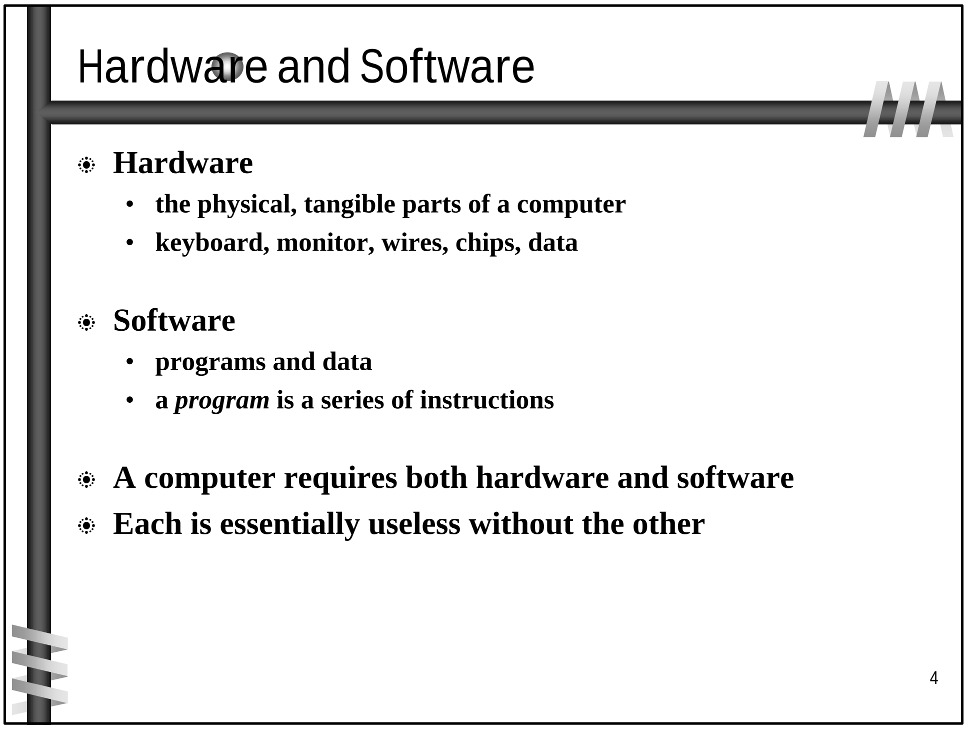
- **Object-Oriented Software Development**
  - **problem solving**
  - **program design and implementation**
  - **object-oriented concepts**
    - **objects**
    - **classes**
    - **interfaces**
    - **inheritance**
    - **polymorphism**
  - **graphics and Graphical User Interfaces**
  - **the Java programming language**



Computer Systems

- We first need to explore the fundamentals of computer processing
- Chapter 1 focuses on:
  - components of a computer
  - how those components interact
  - how computers store and manipulate information
  - computer networks
  - the Internet and the World-Wide Web
  - programming and programming languages
  - graphic systems

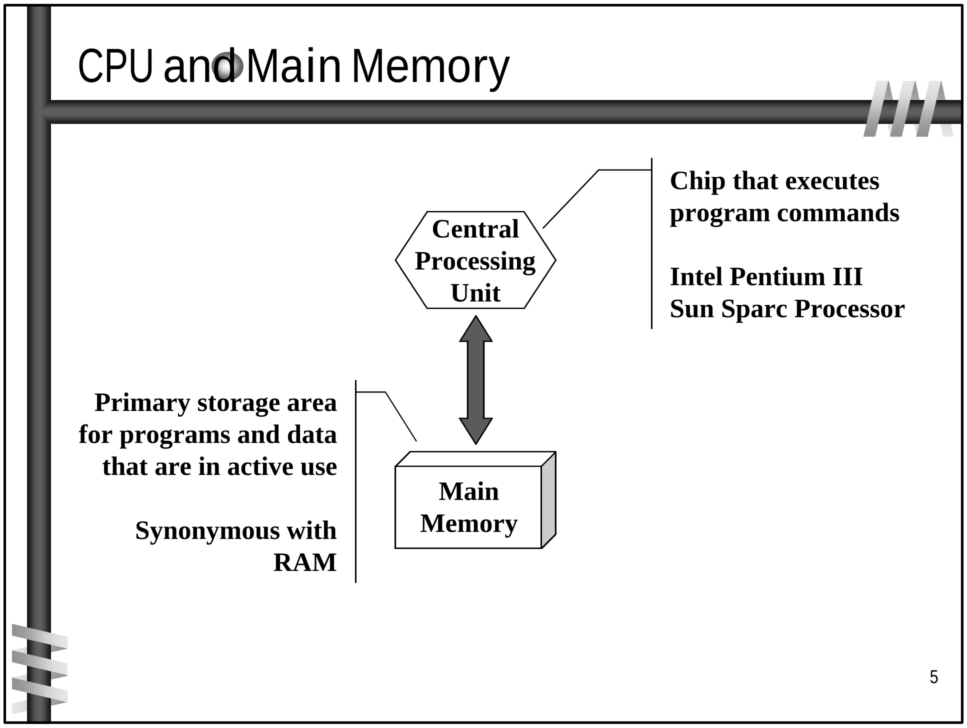
3



Hardware and Software

- Hardware
  - the physical, tangible parts of a computer
  - keyboard, monitor, wires, chips, data
- Software
  - programs and data
  - a *program* is a series of instructions
- A computer requires both hardware and software
- Each is essentially useless without the other

4



CPU and Main Memory

Central Processing Unit

Chip that executes program commands

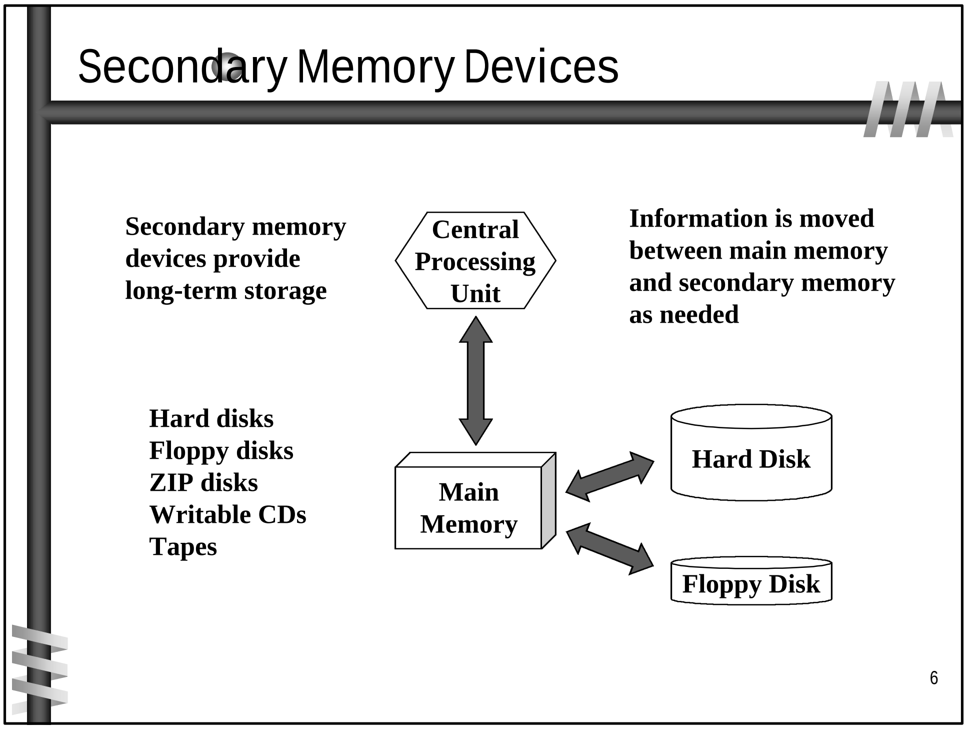
Intel Pentium III
Sun Sparc Processor

Main Memory

Primary storage area for programs and data that are in active use

Synonymous with RAM

5



Secondary Memory Devices

Secondary memory devices provide long-term storage

Central Processing Unit

Information is moved between main memory and secondary memory as needed

Hard disks
Floppy disks
ZIP disks
Writable CDs
Tapes

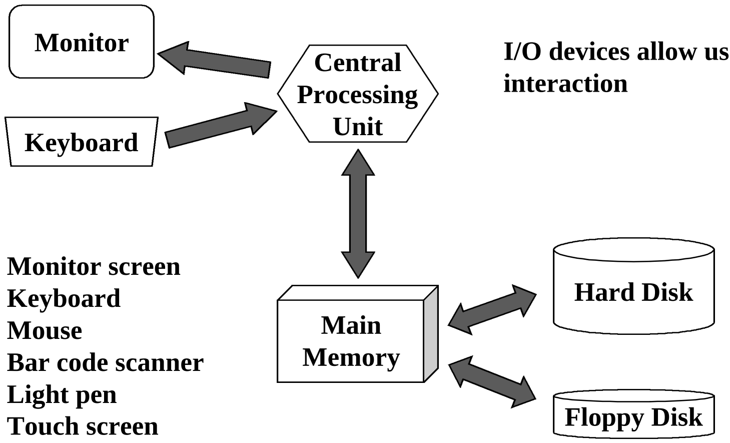
Main Memory

Hard Disk

Floppy Disk

6

Input / Output Devices

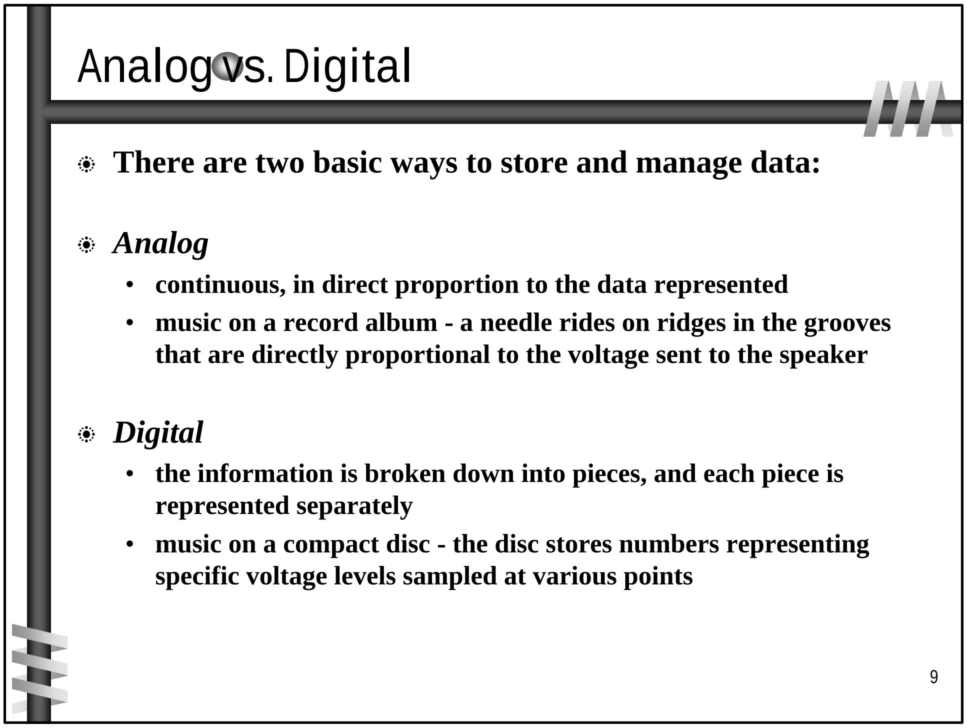


I/O devices allow user interaction

Monitor screen
Keyboard
Mouse
Bar code scanner
Light pen
Touch screen

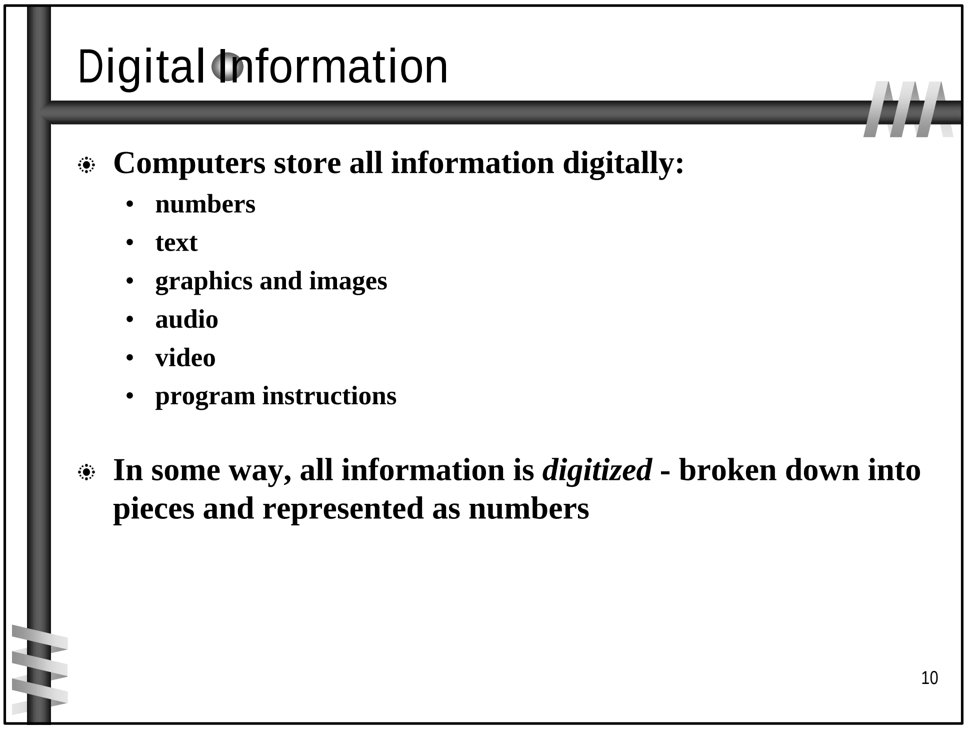
Software Categories

- **Operating System**
  - controls all machine activities
  - provides the user interface to the computer
  - manages resources such as the CPU and memory
  - Windows 98, Windows NT, Unix, Linux, Mac OS
- **Application program**
  - generic term for any other kind of software
  - word processors, missile control systems, games
- **Most operating systems and application programs have a graphical user interface (GUI)**



Analog vs. Digital

- **There are two basic ways to store and manage data:**
- ***Analog***
  - **continuous, in direct proportion to the data represented**
  - **music on a record album - a needle rides on ridges in the grooves that are directly proportional to the voltage sent to the speaker**
- ***Digital***
  - **the information is broken down into pieces, and each piece is represented separately**
  - **music on a compact disc - the disc stores numbers representing specific voltage levels sampled at various points**



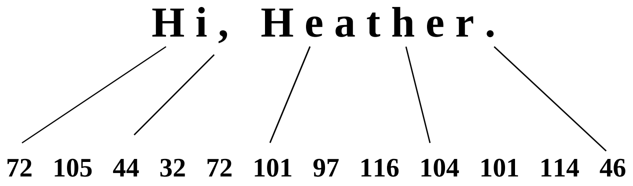
Digital Information

- **Computers store all information digitally:**
  - **numbers**
  - **text**
  - **graphics and images**
  - **audio**
  - **video**
  - **program instructions**
- **In some way, all information is *digitized* - broken down into pieces and represented as numbers**

Representing Text Digitally

- For example, every character is stored as a number, including spaces, digits, and punctuation

- Corresponding upper and lower case letters are separate characters



Binary Numbers

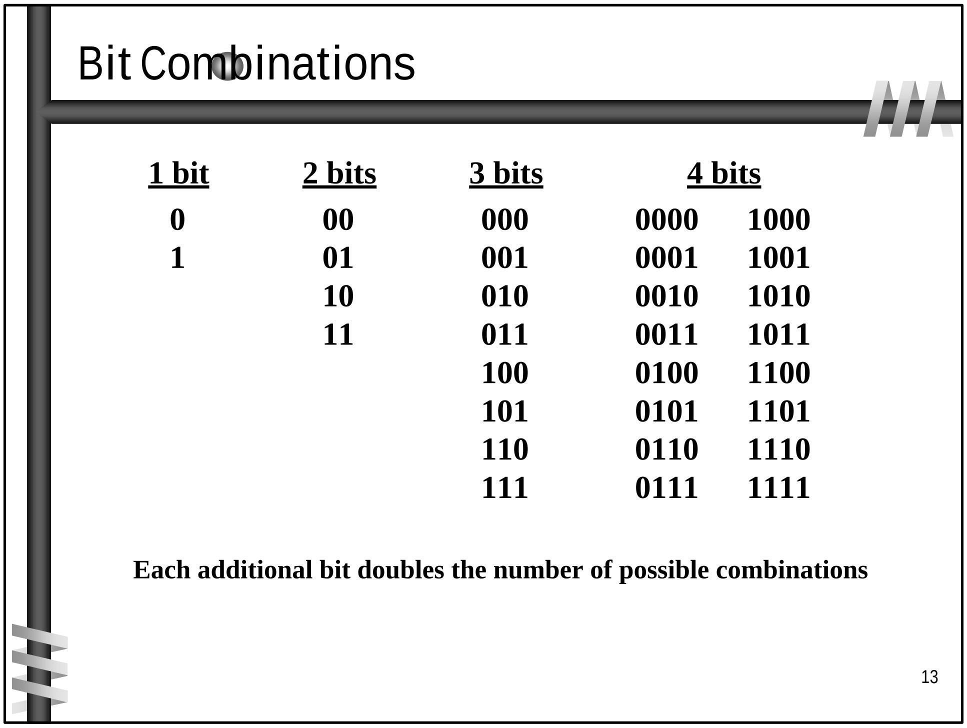
- Once information is digitized, it is represented and stored in memory using the *binary number system*

- A single binary digit (0 or 1) is called a *bit*

- Devices that store and move information are cheaper and more reliable if they only have to represent two states

- A single bit can represent two possible states, like a light bulb that is either on (1) or off (0)

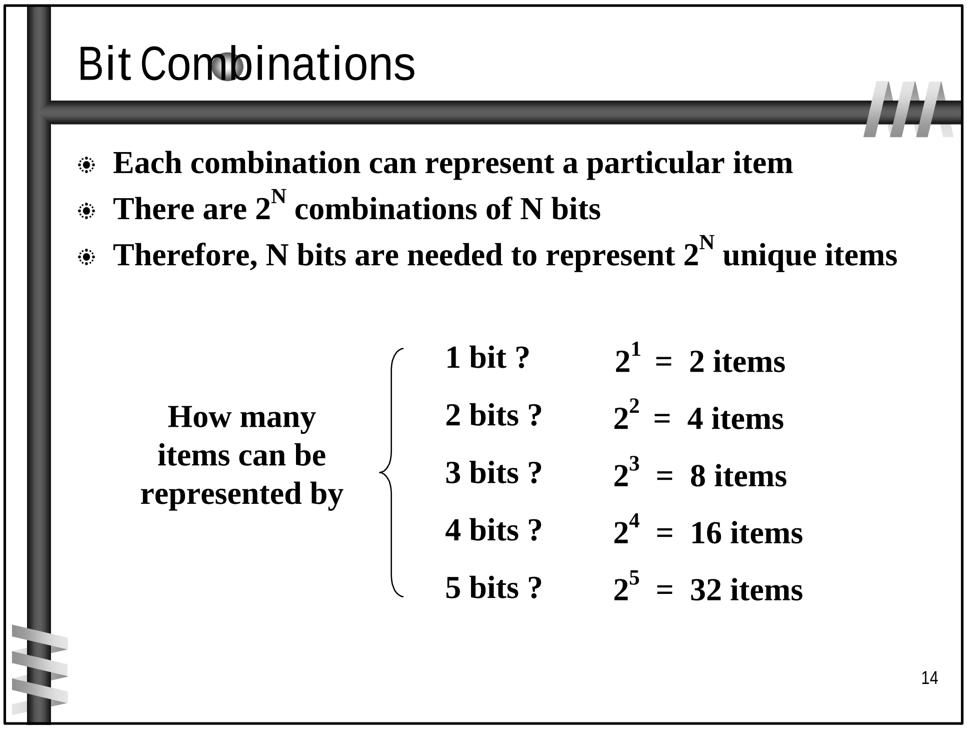
- Combinations of bits are used to store values



Bit Combinations

1 bit	2 bits	3 bits	4 bits	
0	00	000	0000	1000
1	01	001	0001	1001
	10	010	0010	1010
	11	011	0011	1011
		100	0100	1100
		101	0101	1101
		110	0110	1110
		111	0111	1111

Each additional bit doubles the number of possible combinations

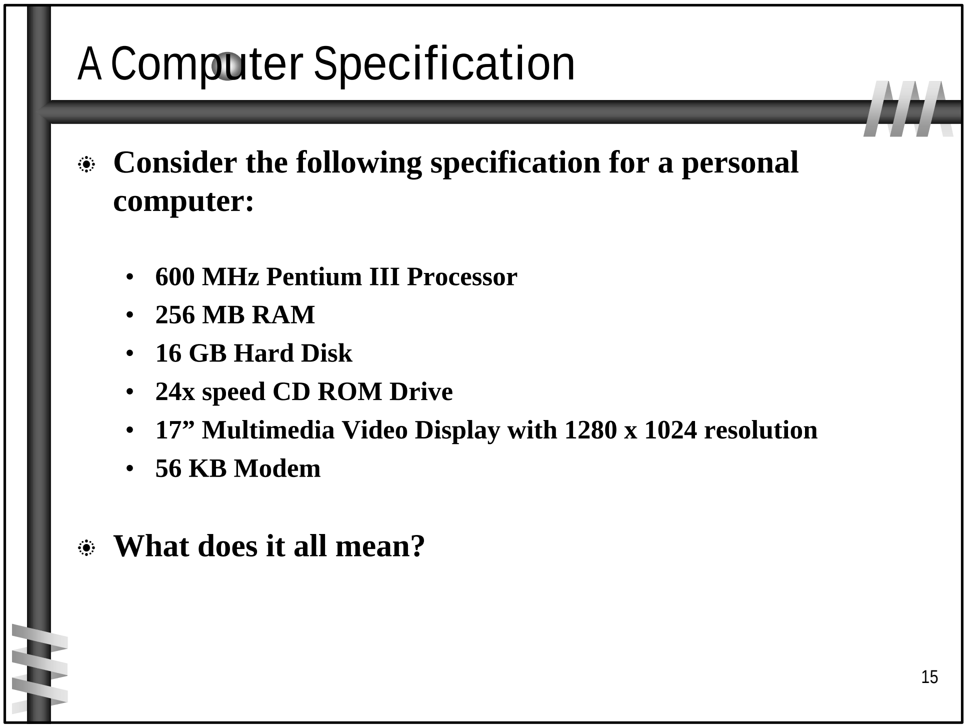


Bit Combinations

- **Each combination can represent a particular item**
- **There are 2^N combinations of N bits**
- **Therefore, N bits are needed to represent 2^N unique items**

How many items can be represented by

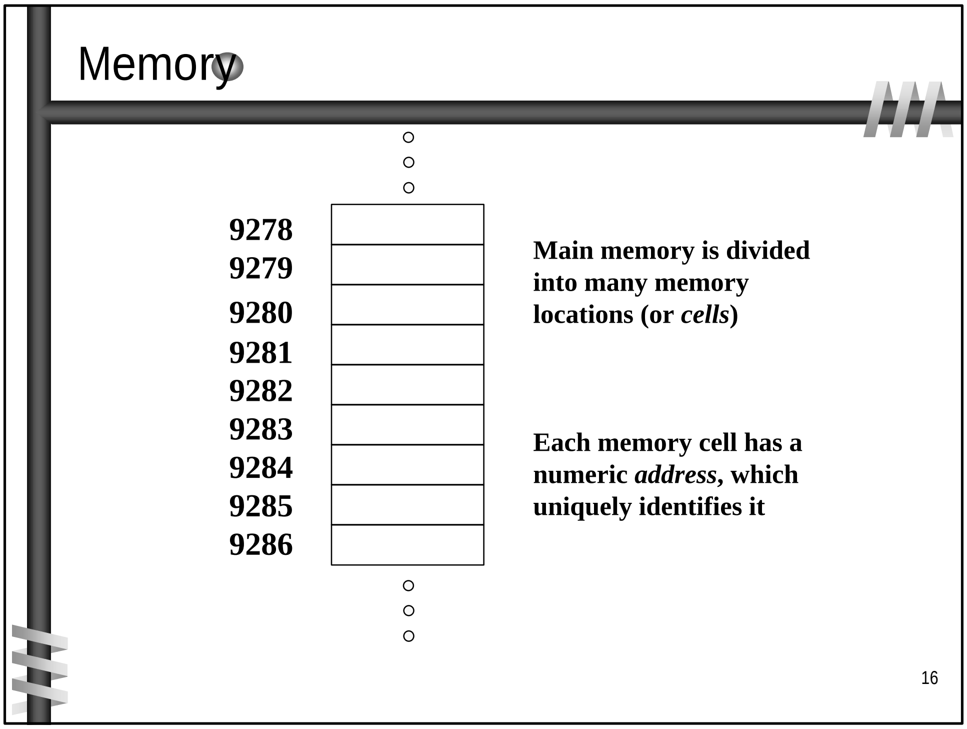
1 bit ?	**2^1 = 2 items**
2 bits ?	**2^2 = 4 items**
3 bits ?	**2^3 = 8 items**
4 bits ?	**2^4 = 16 items**
5 bits ?	**2^5 = 32 items**



A Computer Specification

- Consider the following specification for a personal computer:
  - 600 MHz Pentium III Processor
  - 256 MB RAM
  - 16 GB Hard Disk
  - 24x speed CD ROM Drive
  - 17” Multimedia Video Display with 1280 x 1024 resolution
  - 56 KB Modem
- What does it all mean?

15



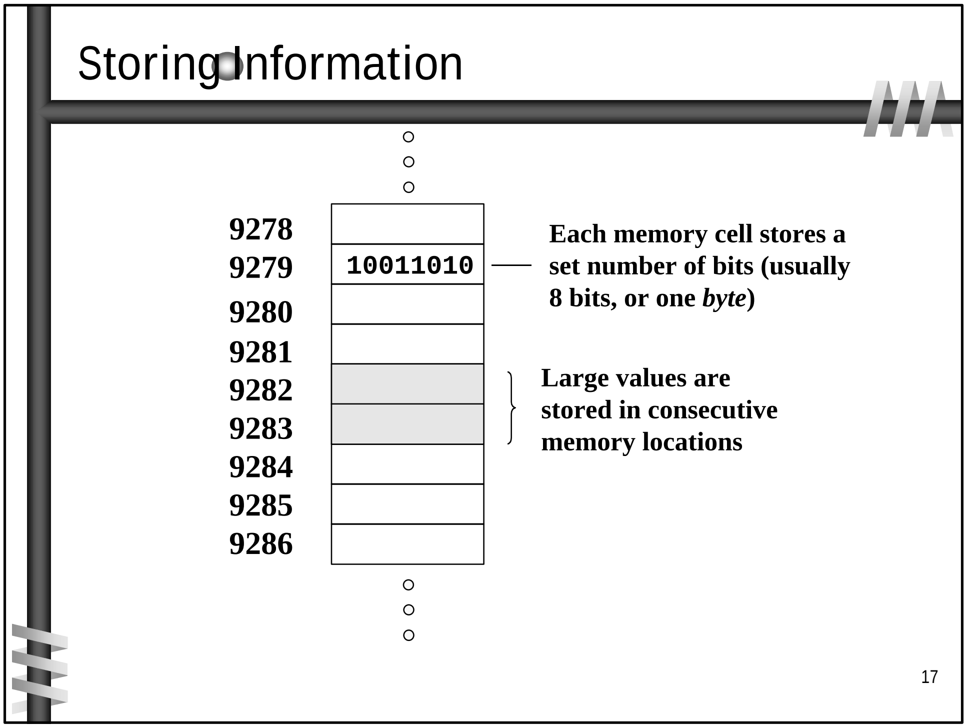
Memory

9278
9279
9280
9281
9282
9283
9284
9285
9286

Main memory is divided into many memory locations (or *cells*)

Each memory cell has a numeric *address*, which uniquely identifies it

16

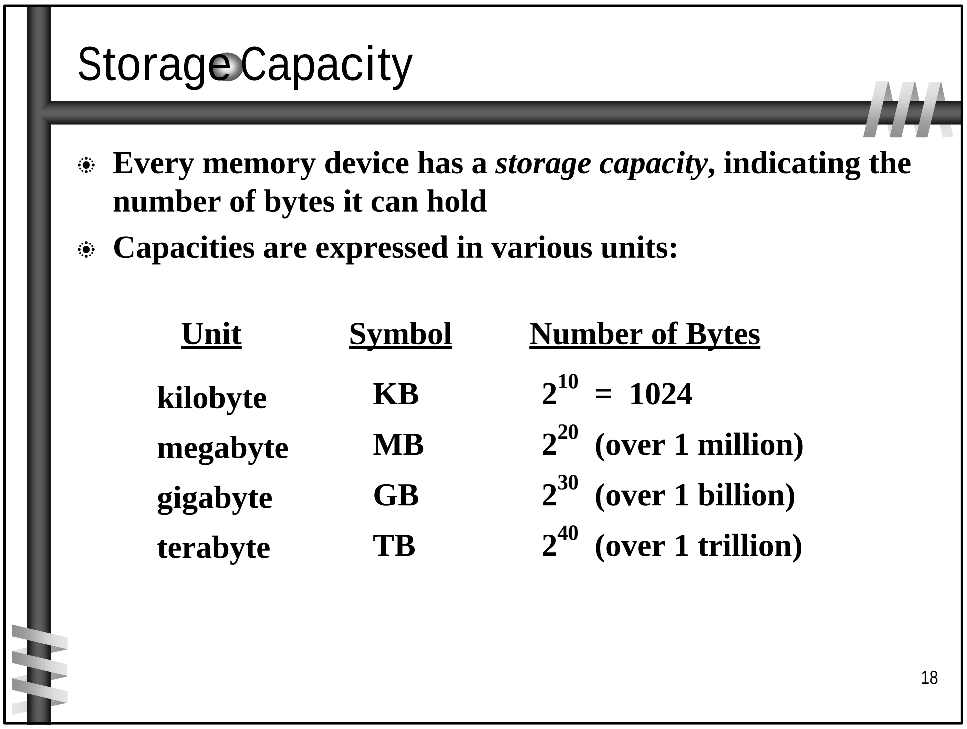


Storing Information

9278
9279 10011010
9280
9281
9282
9283
9284
9285
9286

Each memory cell stores a set number of bits (usually 8 bits, or one *byte*)

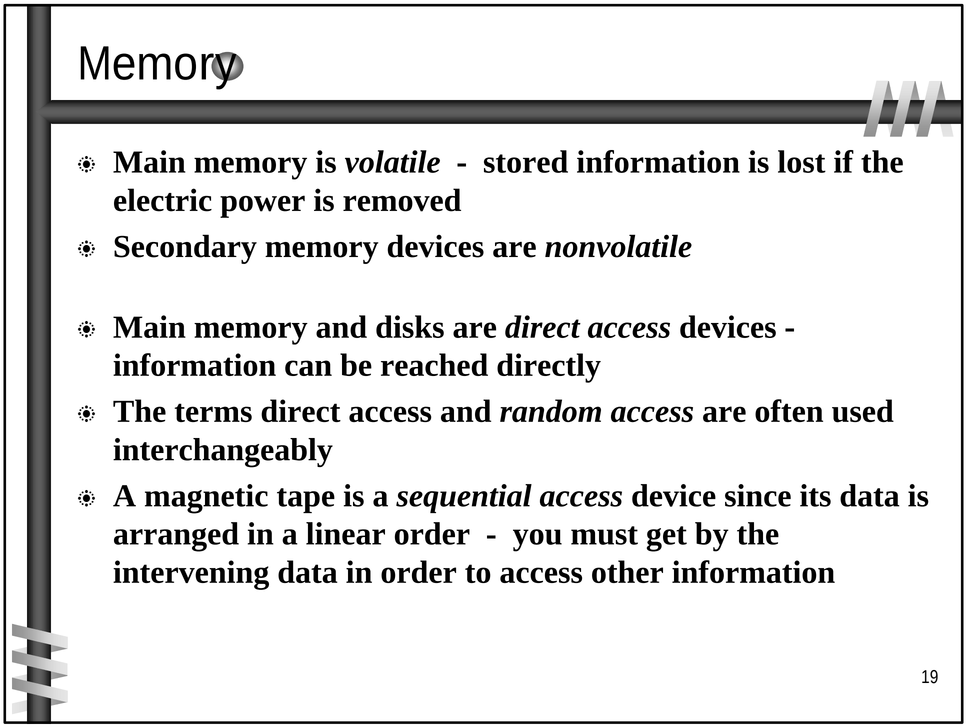
Large values are stored in consecutive memory locations



Storage Capacity

- **Every memory device has a *storage capacity*, indicating the number of bytes it can hold**
- **Capacities are expressed in various units:**

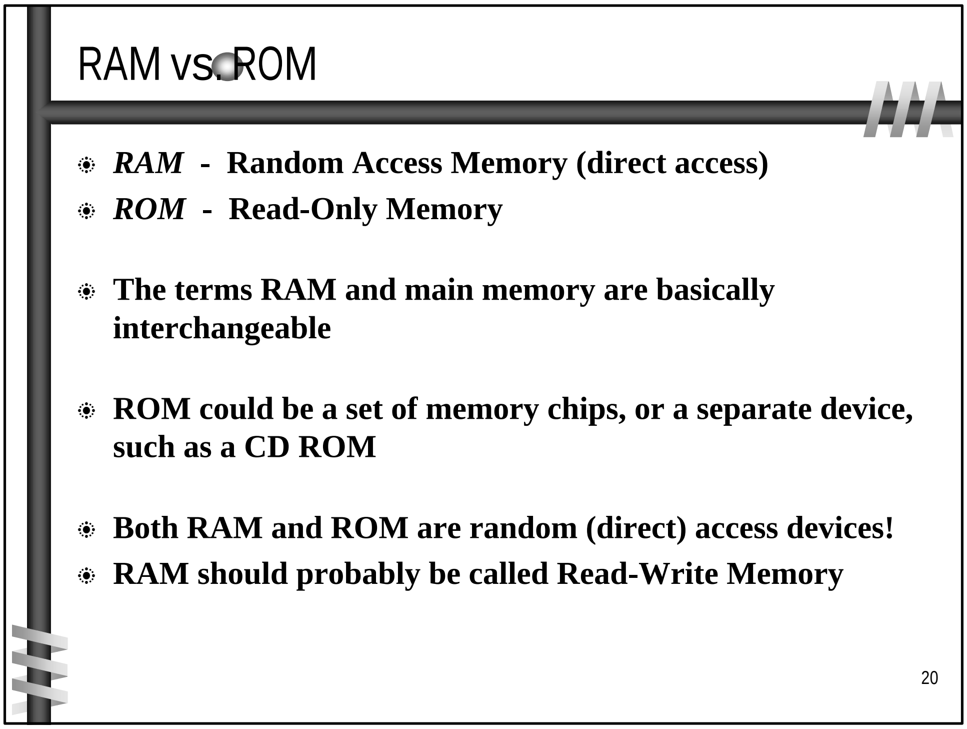
Unit	Symbol	Number of Bytes
kilobyte	**KB**	**2^{10} = 1024**
megabyte	**MB**	**2^{20} (over 1 million)**
gigabyte	**GB**	**2^{30} (over 1 billion)**
terabyte	**TB**	**2^{40} (over 1 trillion)**



Memory

- Main memory is *volatile* - stored information is lost if the electric power is removed
- Secondary memory devices are *nonvolatile*

- Main memory and disks are *direct access* devices - information can be reached directly
- The terms direct access and *random access* are often used interchangeably
- A magnetic tape is a *sequential access* device since its data is arranged in a linear order - you must get by the intervening data in order to access other information



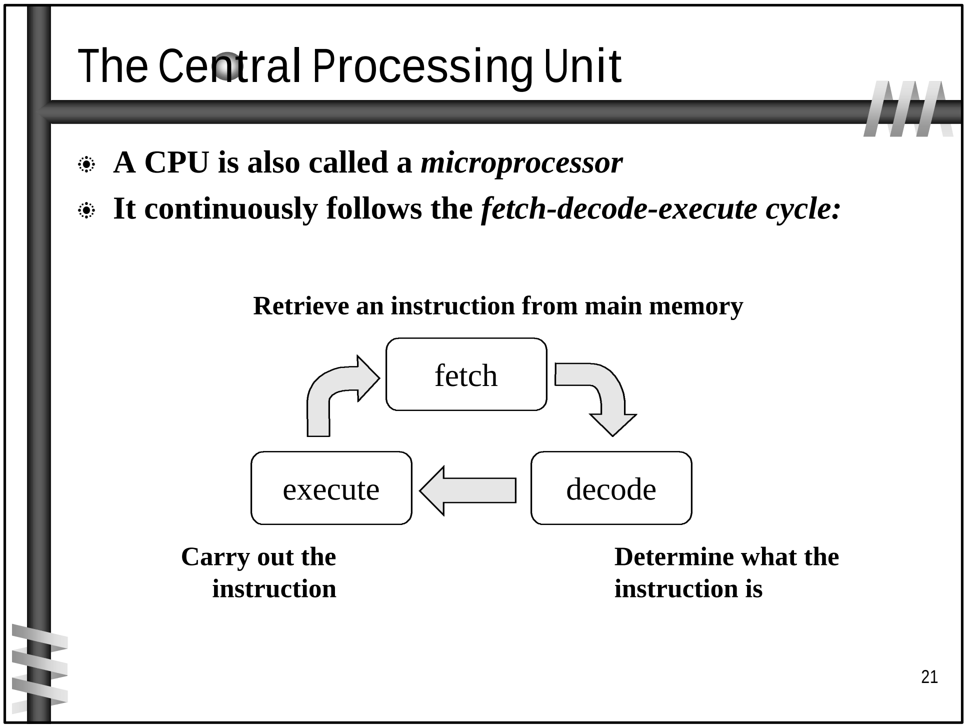
RAM vs. ROM

- *RAM* - Random Access Memory (direct access)
- *ROM* - Read-Only Memory

- The terms RAM and main memory are basically interchangeable

- ROM could be a set of memory chips, or a separate device, such as a CD ROM

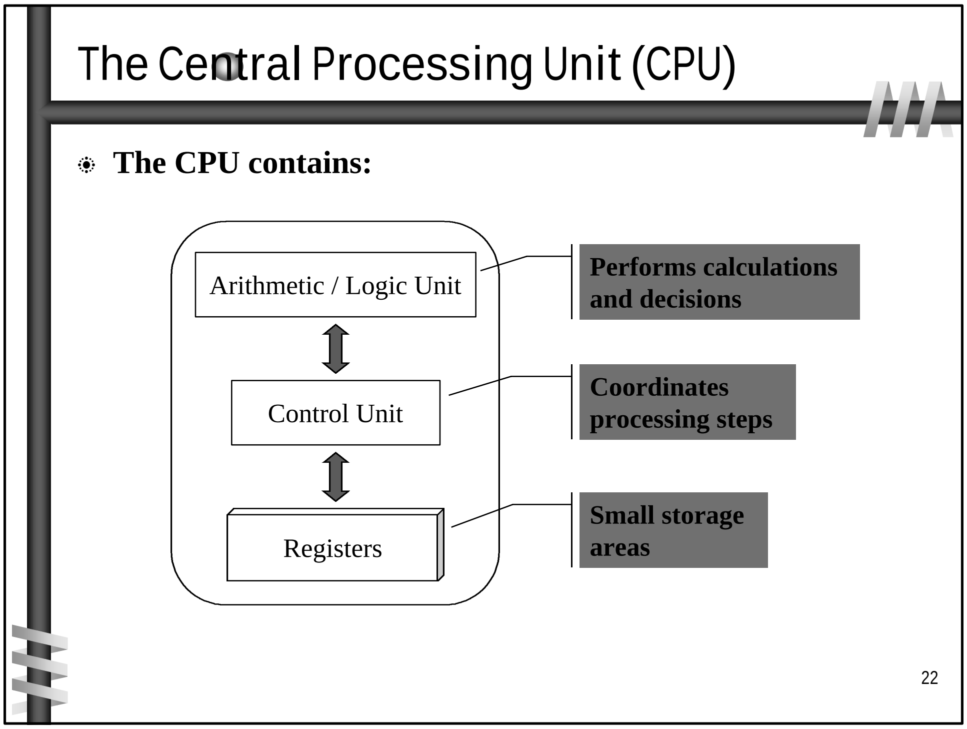
- Both RAM and ROM are random (direct) access devices!
- RAM should probably be called Read-Write Memory



The Central Processing Unit

- A CPU is also called a *microprocessor*
- It continuously follows the *fetch-decode-execute cycle:*

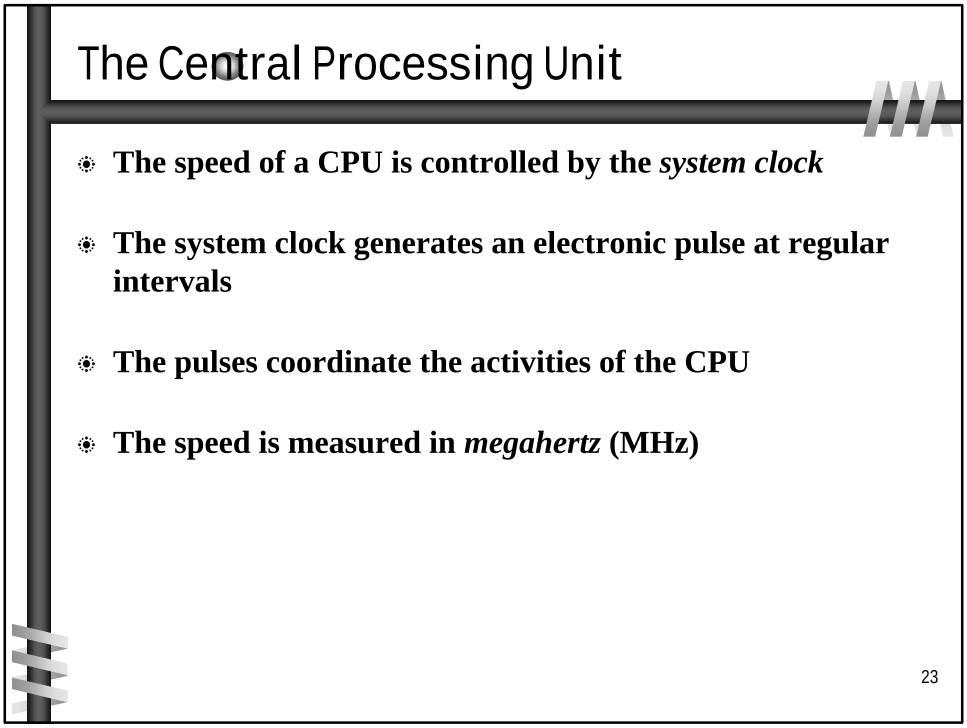
Retrieve an instruction from main memory
fetch
decode
execute
Carry out the instruction
Determine what the instruction is



The Central Processing Unit (CPU)

- The CPU contains:

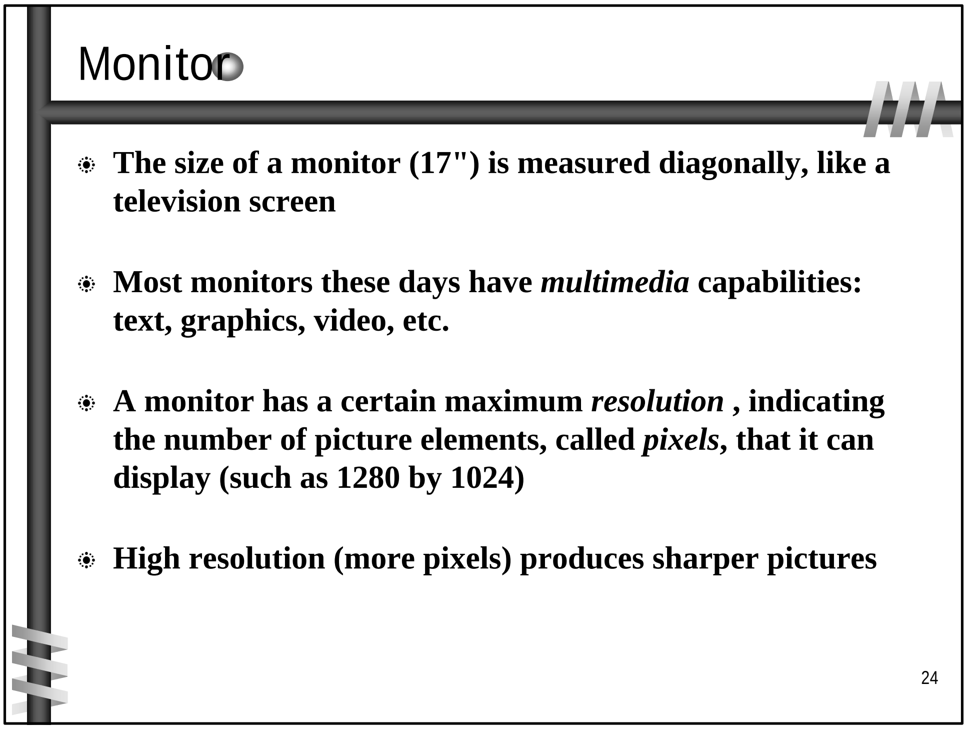
Arithmetic / Logic Unit
Control Unit
Registers
Performs calculations and decisions
Coordinates processing steps
Small storage areas



The Central Processing Unit

- **The speed of a CPU is controlled by the *system clock***
- **The system clock generates an electronic pulse at regular intervals**
- **The pulses coordinate the activities of the CPU**
- **The speed is measured in *megahertz* (MHz)**

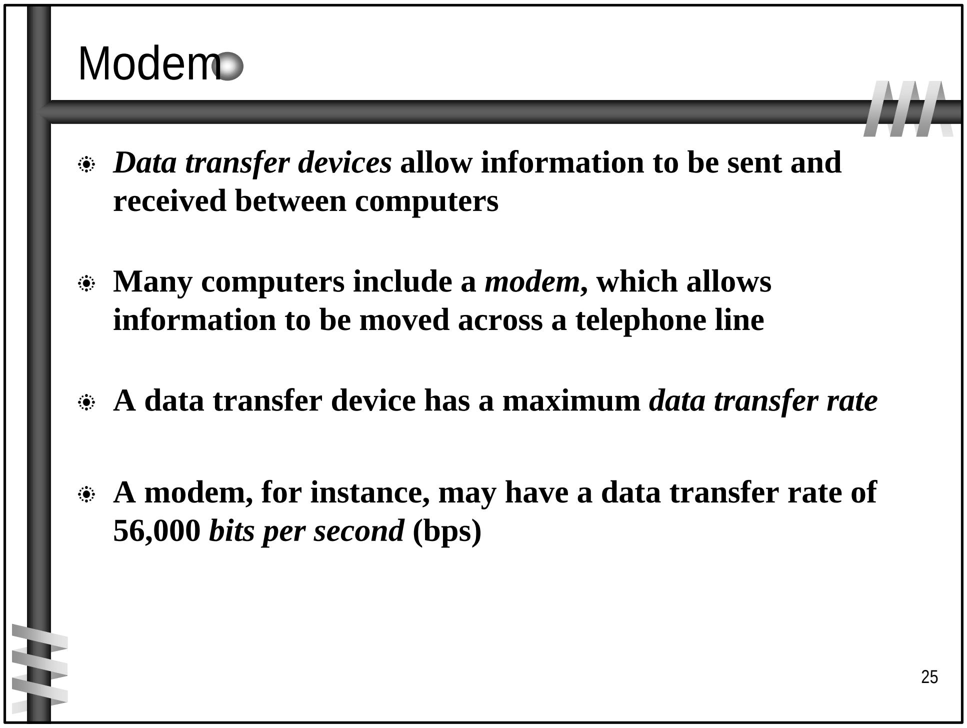
23



Monitor

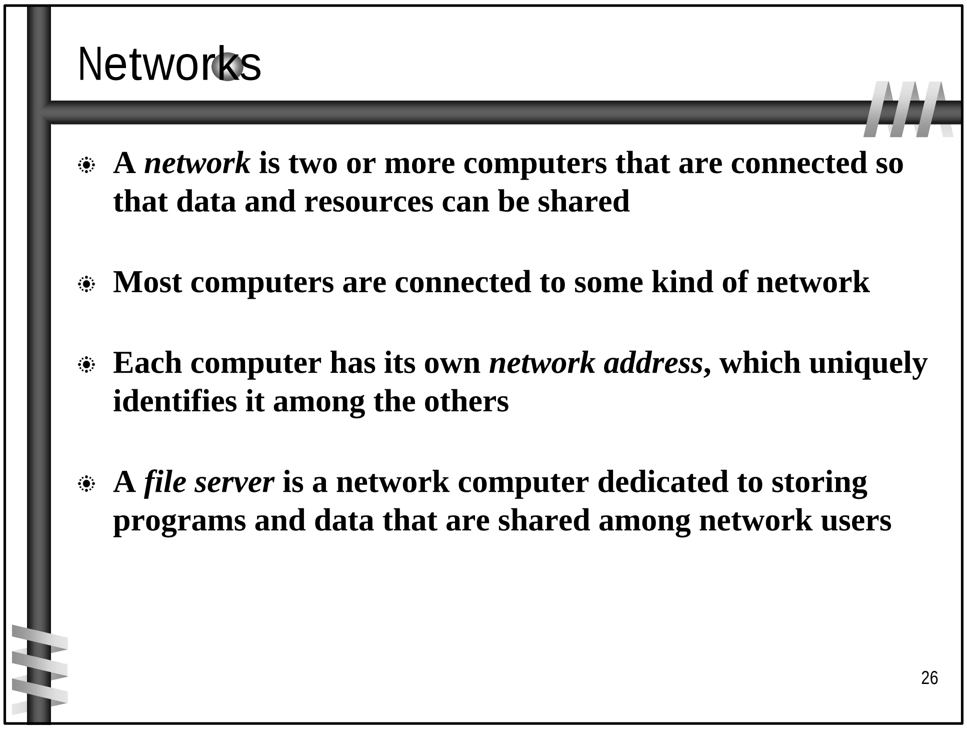
- **The size of a monitor (17") is measured diagonally, like a television screen**
- **Most monitors these days have *multimedia* capabilities: text, graphics, video, etc.**
- **A monitor has a certain maximum *resolution* , indicating the number of picture elements, called *pixels*, that it can display (such as 1280 by 1024)**
- **High resolution (more pixels) produces sharper pictures**

24



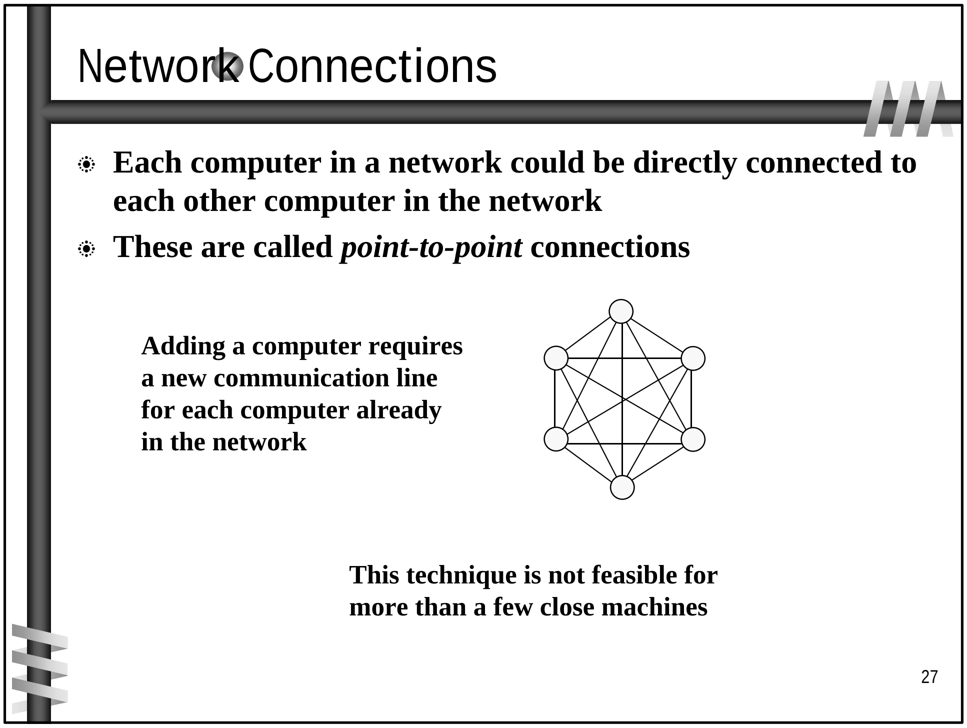
Modem

- *Data transfer devices* allow information to be sent and received between computers
- Many computers include a *modem,* which allows information to be moved across a telephone line
- A data transfer device has a maximum *data transfer rate*
- A modem, for instance, may have a data transfer rate of 56,000 *bits per second* (bps)



Networks

- A *network* is two or more computers that are connected so that data and resources can be shared
- Most computers are connected to some kind of network
- Each computer has its own *network address,* which uniquely identifies it among the others
- A *file server* is a network computer dedicated to storing programs and data that are shared among network users

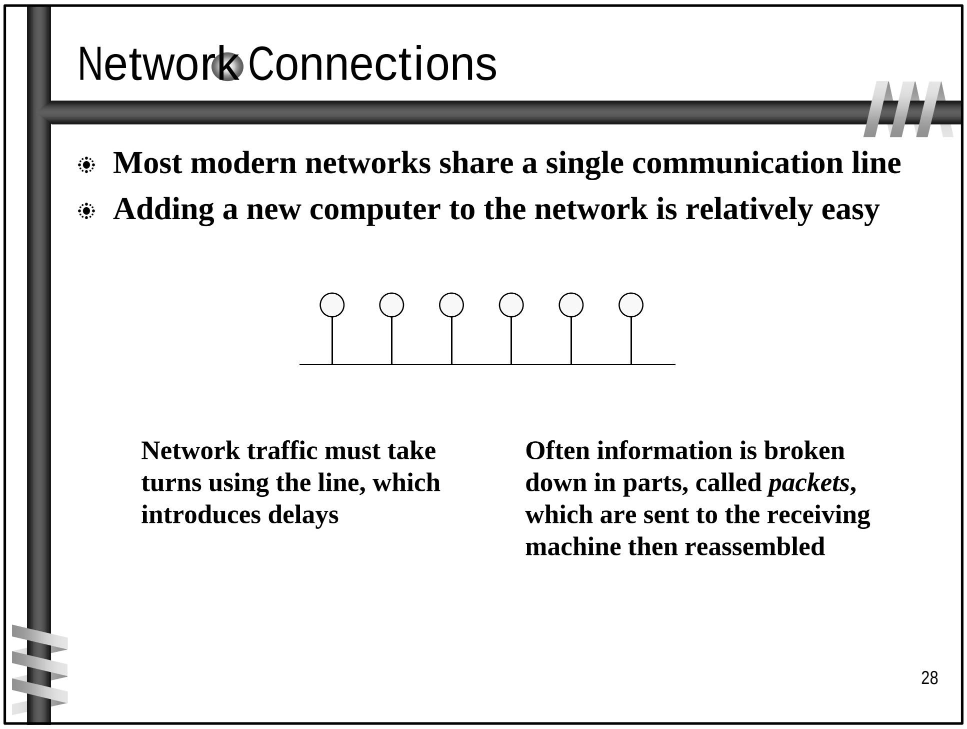


Network Connections

- Each computer in a network could be directly connected to each other computer in the network
- These are called *point-to-point* connections

Adding a computer requires a new communication line for each computer already in the network

This technique is not feasible for more than a few close machines



Network Connections

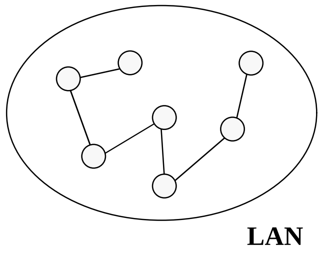
- Most modern networks share a single communication line
- Adding a new computer to the network is relatively easy

Network traffic must take turns using the line, which introduces delays

Often information is broken down in parts, called *packets*, which are sent to the receiving machine then reassembled

Local-Area Networks

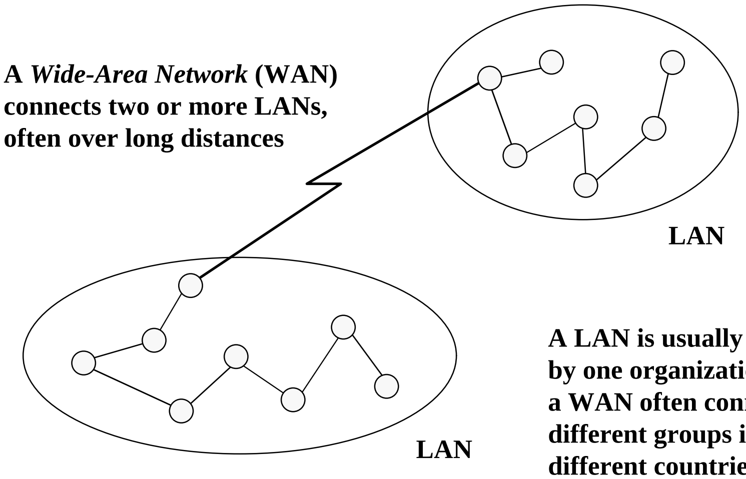
A *Local-Area Network* (LAN) covers a small distance and a small number of computers



A LAN often connects the machines in a single room or building

Wide-Area Networks

A *Wide-Area Network* (WAN) connects two or more LANs, often over long distances



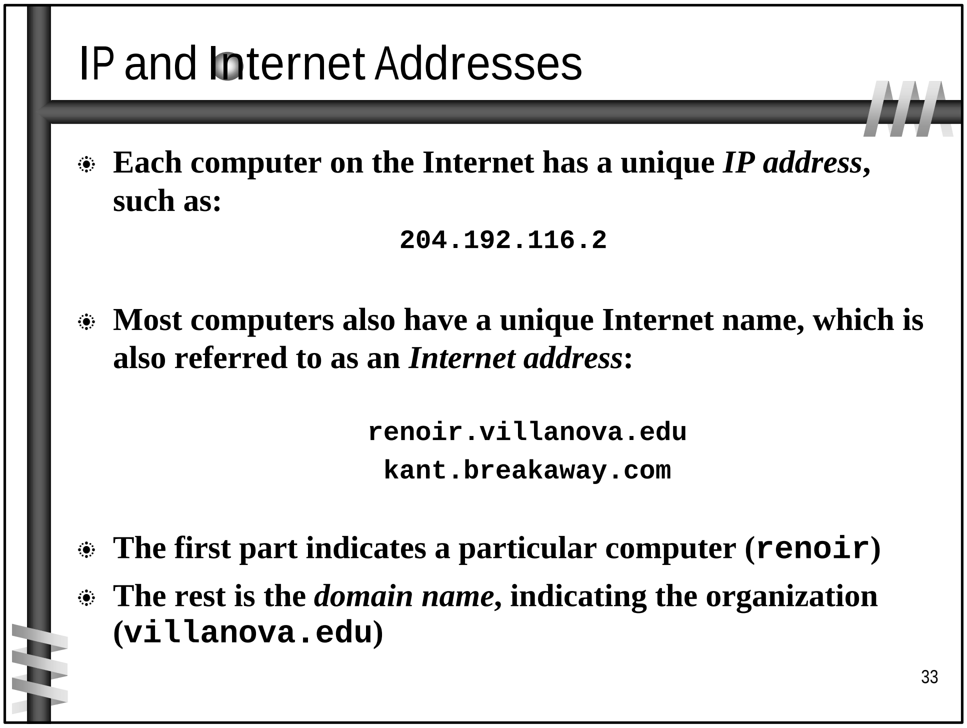
A LAN is usually owned by one organization, but a WAN often connects different groups in different countries

The Internet

- The *Internet* is a WAN which spans the entire planet
- The word Internet comes from the term *internetworking*, which implies communication among networks
- It started as a United States government project, sponsored by the Advanced Research Projects Agency (ARPA), and was originally called the ARPANET
- The Internet grew quickly throughout the 1980s and 90s
- Less than 600 computers were connected to the Internet in 1983; now there are over 10 million

TCP/IP

- A protocol is a set of rules that determine how things communicate with each other
- The software which manages Internet communication follows a suite of protocols called *TCP/IP*
- The *Internet Protocol* (IP) determines the format of the information as it is transferred
- The *Transmission Control Protocol* (TCP) dictates how messages are reassembled and handles lost information



IP and Internet Addresses

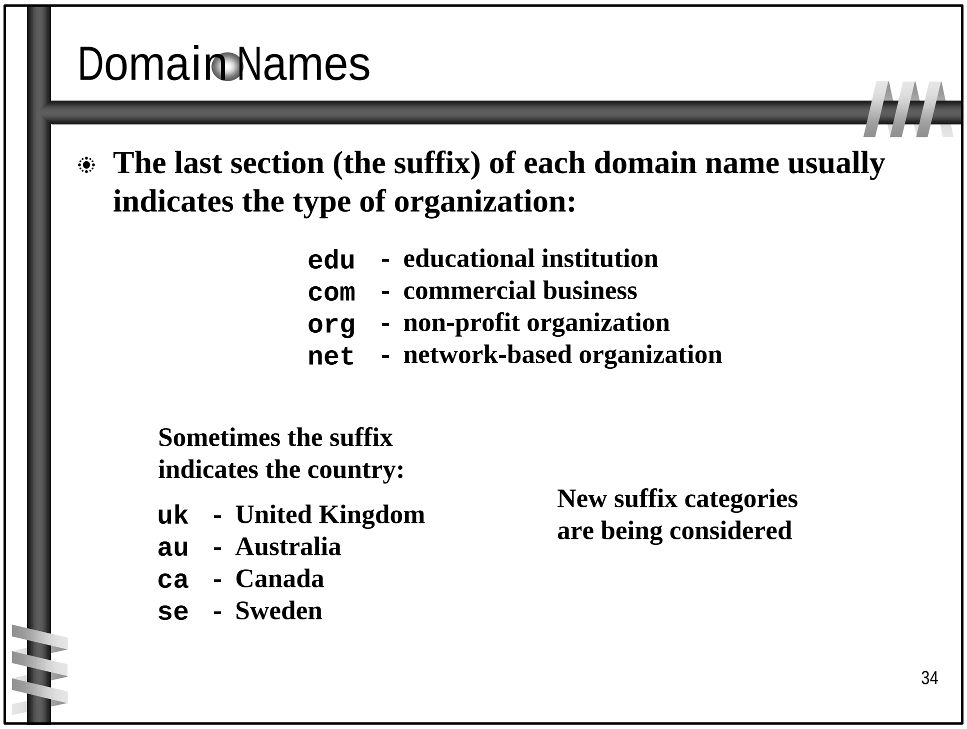
- **Each computer on the Internet has a unique *IP address*, such as:**

```
204.192.116.2
```

- **Most computers also have a unique Internet name, which is also referred to as an *Internet address*:**

```
renoir.villanova.edu
kant.breakaway.com
```

- **The first part indicates a particular computer (`renoir`)**
- **The rest is the *domain name*, indicating the organization (`villanova.edu`)**



Domain Names

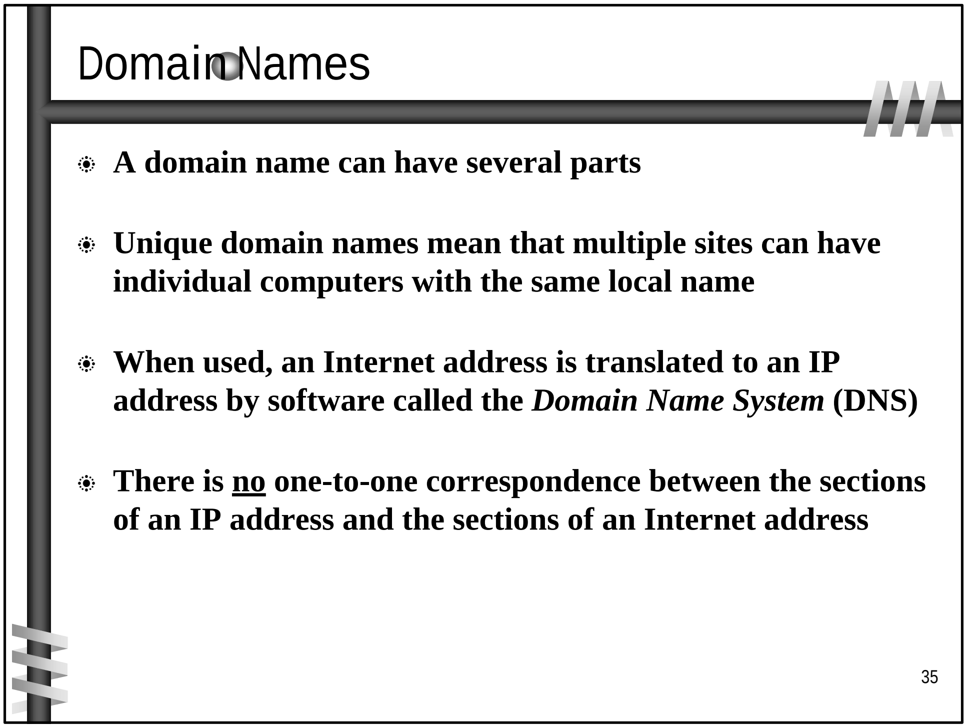
- **The last section (the suffix) of each domain name usually indicates the type of organization:**

`edu`	-	**educational institution**
`com`	-	**commercial business**
`org`	-	**non-profit organization**
`net`	-	**network-based organization**

Sometimes the suffix indicates the country:

`uk`	-	**United Kingdom**
`au`	-	**Australia**
`ca`	-	**Canada**
`se`	-	**Sweden**

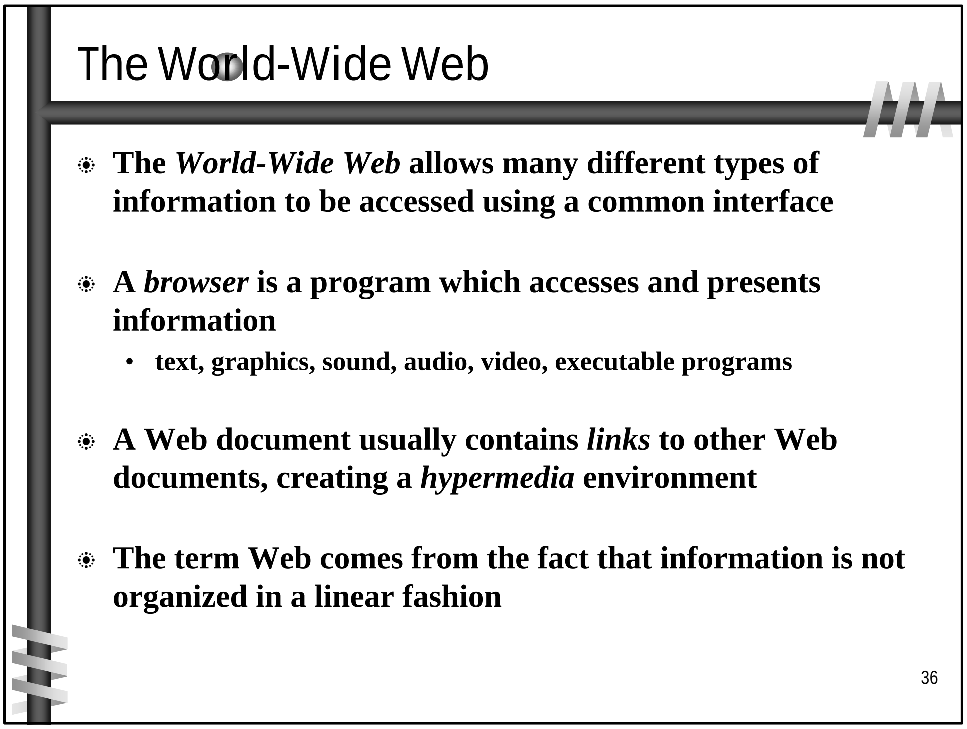
New suffix categories are being considered



Domain Names

- **A domain name can have several parts**
- **Unique domain names mean that multiple sites can have individual computers with the same local name**
- **When used, an Internet address is translated to an IP address by software called the *Domain Name System* (DNS)**
- **There is <u>no</u> one-to-one correspondence between the sections of an IP address and the sections of an Internet address**

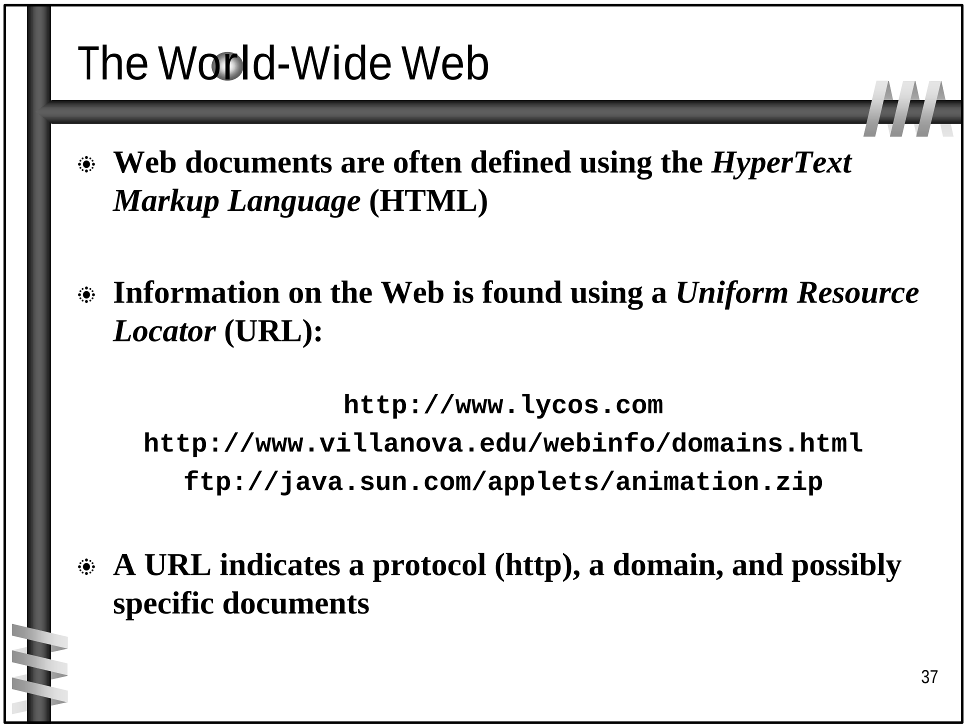
35



The World-Wide Web

- **The *World-Wide Web* allows many different types of information to be accessed using a common interface**
- **A *browser* is a program which accesses and presents information**
  - **text, graphics, sound, audio, video, executable programs**
- **A Web document usually contains *links* to other Web documents, creating a *hypermedia* environment**
- **The term Web comes from the fact that information is not organized in a linear fashion**

36

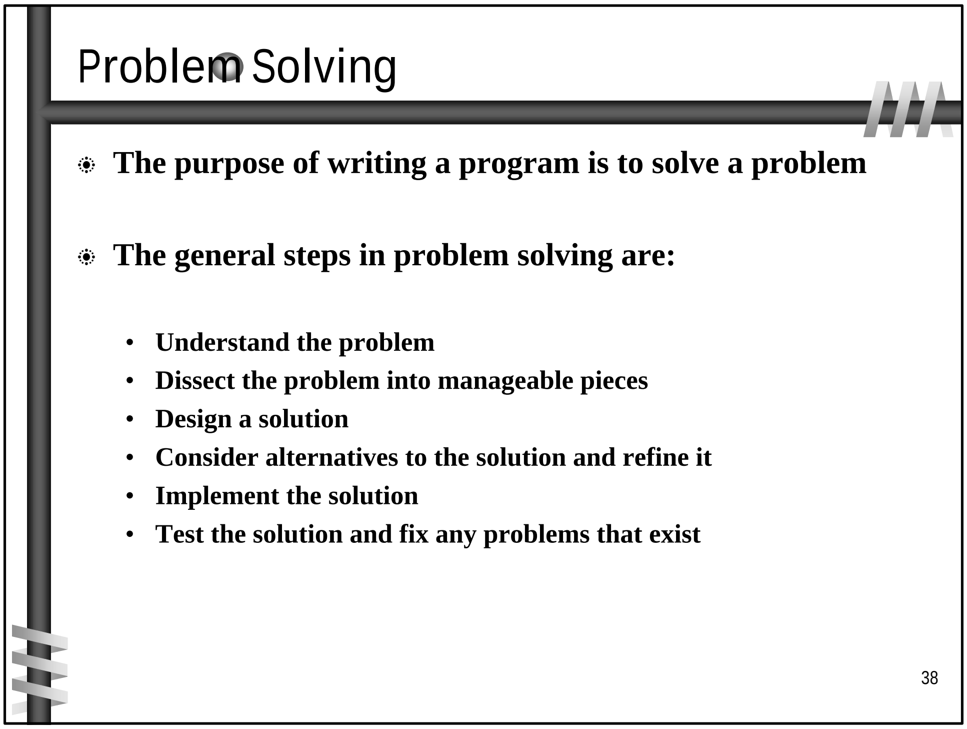


The World-Wide Web

- Web documents are often defined using the *HyperText Markup Language* (HTML)
- Information on the Web is found using a *Uniform Resource Locator* (URL):

```
http://www.lycos.com
http://www.villanova.edu/webinfo/domains.html
ftp://java.sun.com/applets/animation.zip
```

- A URL indicates a protocol (http), a domain, and possibly specific documents



Problem Solving

- The purpose of writing a program is to solve a problem
- The general steps in problem solving are:
  - Understand the problem
  - Dissect the problem into manageable pieces
  - Design a solution
  - Consider alternatives to the solution and refine it
  - Implement the solution
  - Test the solution and fix any problems that exist

Problem Solving

- Many software projects fail because the developer didn't really understand the problem to be solved
- We must avoid assumptions and clarify ambiguities

- As problems and their solutions become larger, we must organize our development into manageable pieces
- This technique is fundamental to software development
- We will dissect our solutions into pieces called classes and objects, taking an *object-oriented approach*

The Java Programming Language

- A *programming language* specifies the words and symbols that we can use to write a program

- A programming language employs a set of rules that dictate how the words and symbols can be put together to form valid *program statements*

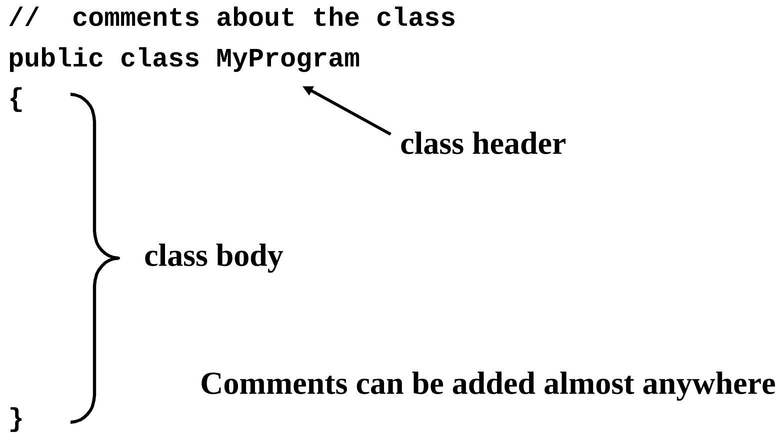
- Java was created by Sun Microsystems, Inc.
- It was introduced in 1995 and has become quite popular
- It is an object-oriented language

Java Program Structure

- In the Java programming language:
  - A program is made up of one or more *classes*
  - A class contains one or more *methods*
  - A method contains program *statements*
- These terms will be explored in detail throughout the course
- A Java application always contains a method called `main`
- See <u>`Lincoln.java`</u> (page 26)

41

Java Program Structure



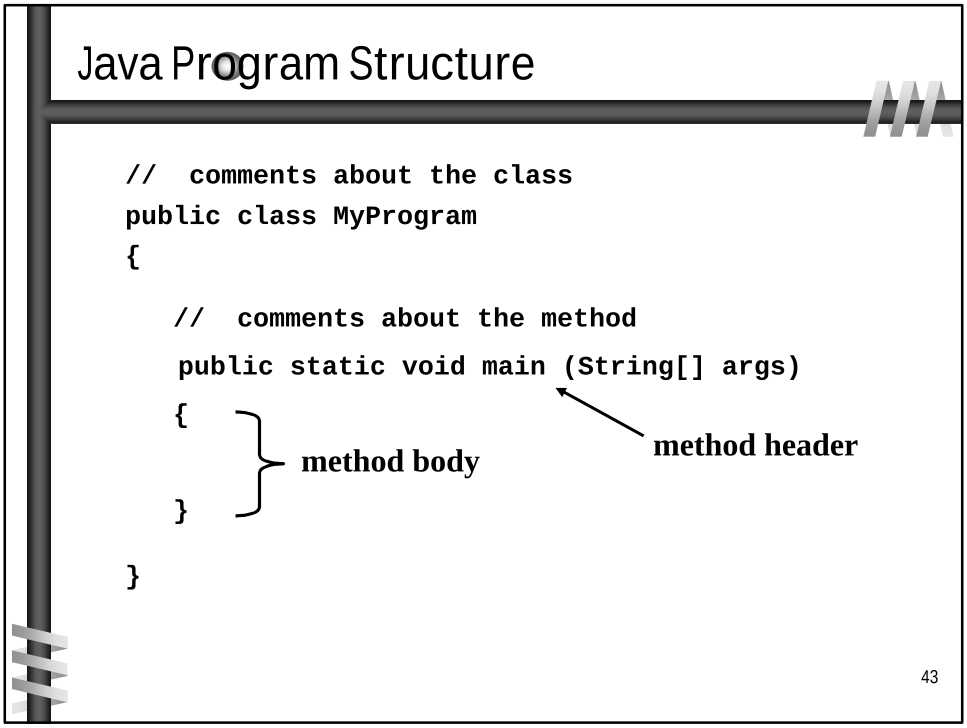
```
//  comments about the class
public class MyProgram
{

}
```

class header

class body

Comments can be added almost anywhere

42

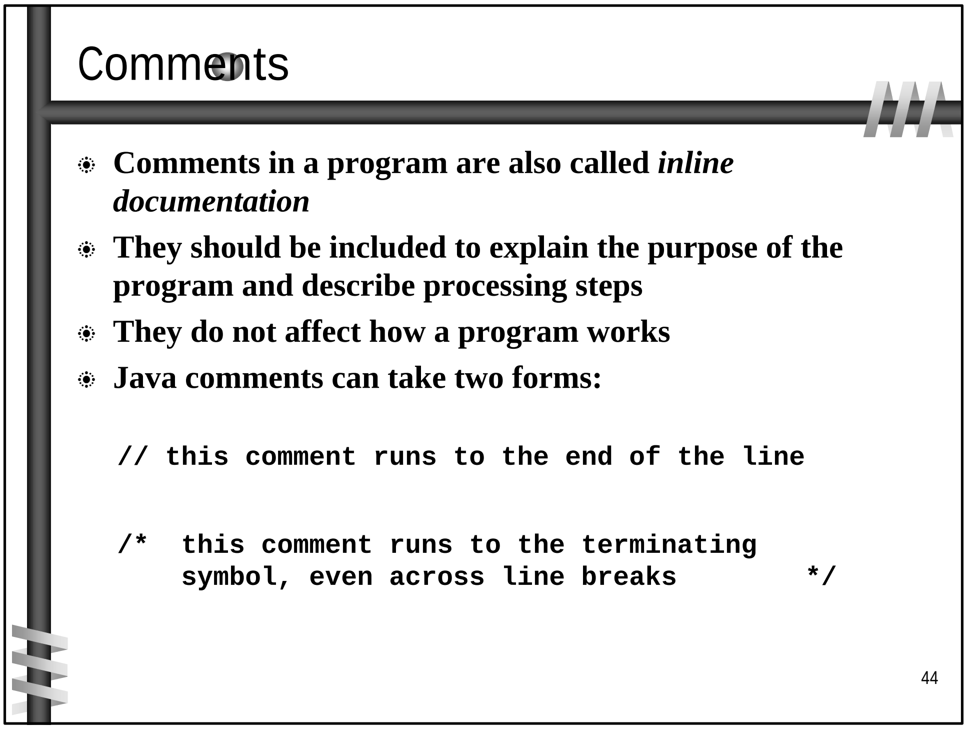


Java Program Structure

```
//  comments about the class
public class MyProgram
{

   //  comments about the method
   public static void main (String[] args)
   {

   }

}
```

method body

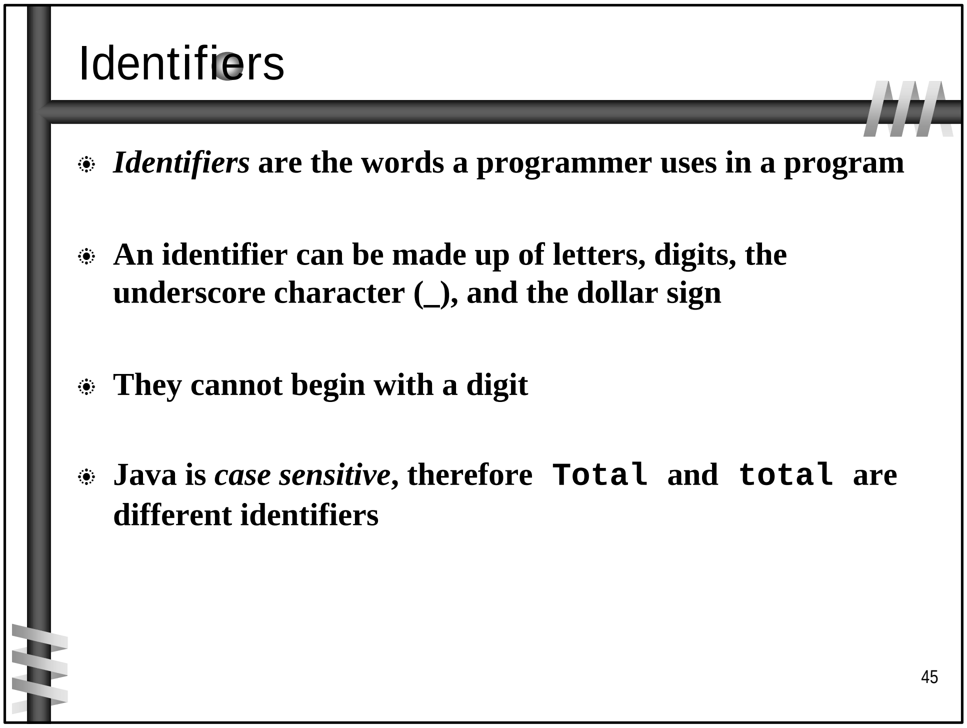
method header



Comments

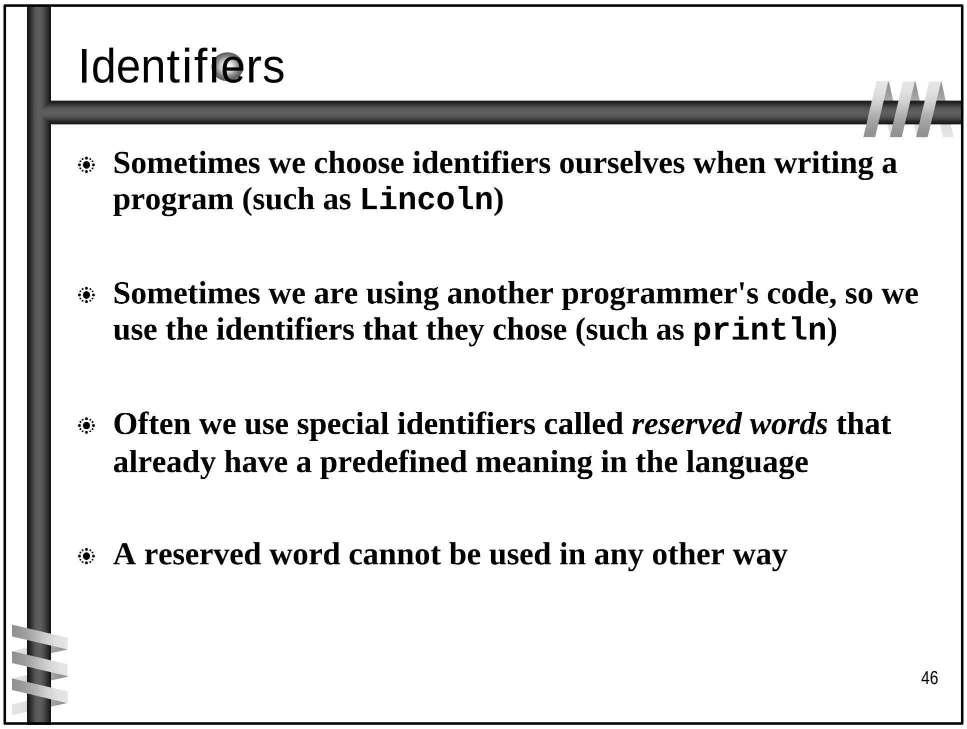
- Comments in a program are also called *inline documentation*
- They should be included to explain the purpose of the program and describe processing steps
- They do not affect how a program works
- Java comments can take two forms:

```
// this comment runs to the end of the line


/*  this comment runs to the terminating
    symbol, even across line breaks        */
```



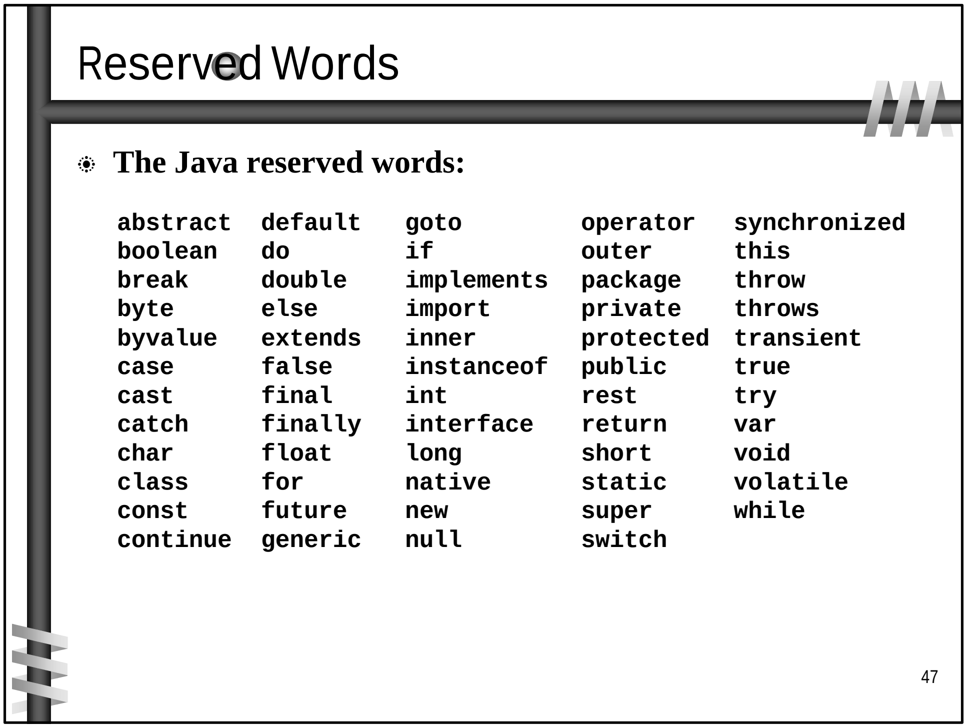
Identifiers

- *Identifiers* are the words a programmer uses in a program
- An identifier can be made up of letters, digits, the underscore character (_), and the dollar sign
- They cannot begin with a digit
- Java is *case sensitive*, therefore `Total` and `total` are different identifiers



Identifiers

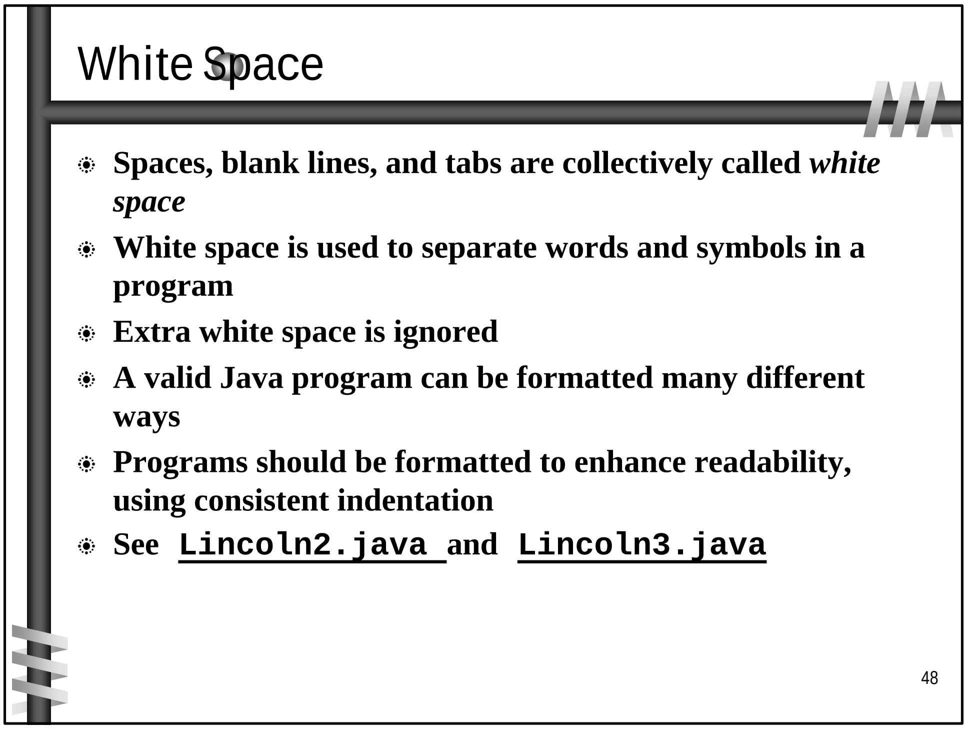
- Sometimes we choose identifiers ourselves when writing a program (such as `Lincoln`)
- Sometimes we are using another programmer's code, so we use the identifiers that they chose (such as `println`)
- Often we use special identifiers called *reserved words* that already have a predefined meaning in the language
- A reserved word cannot be used in any other way



Reserved Words

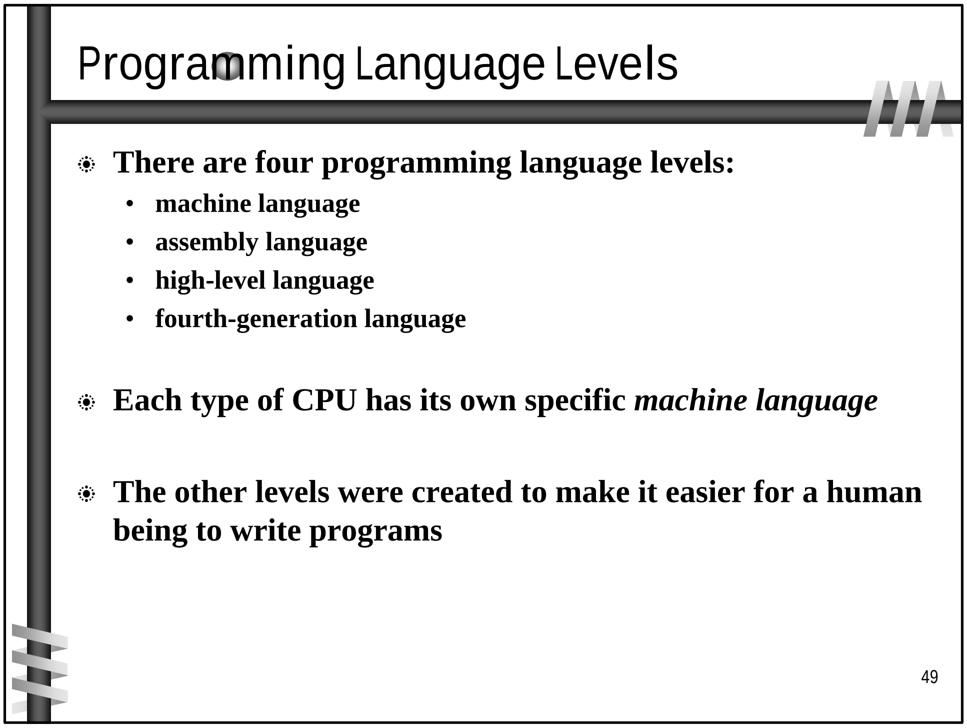
- **The Java reserved words:**

```
abstract  default   goto        operator   synchronized
boolean   do        if          outer      this
break     double    implements  package    throw
byte      else      import      private    throws
byvalue   extends   inner       protected  transient
case      false     instanceof  public     true
cast      final     int         rest       try
catch     finally   interface   return     var
char      float     long        short      void
class     for       native      static     volatile
const     future    new         super      while
continue  generic   null        switch
```



White Space

- **Spaces, blank lines, and tabs are collectively called *white space***
- **White space is used to separate words and symbols in a program**
- **Extra white space is ignored**
- **A valid Java program can be formatted many different ways**
- **Programs should be formatted to enhance readability, using consistent indentation**
- **See `Lincoln2.java` and `Lincoln3.java`**

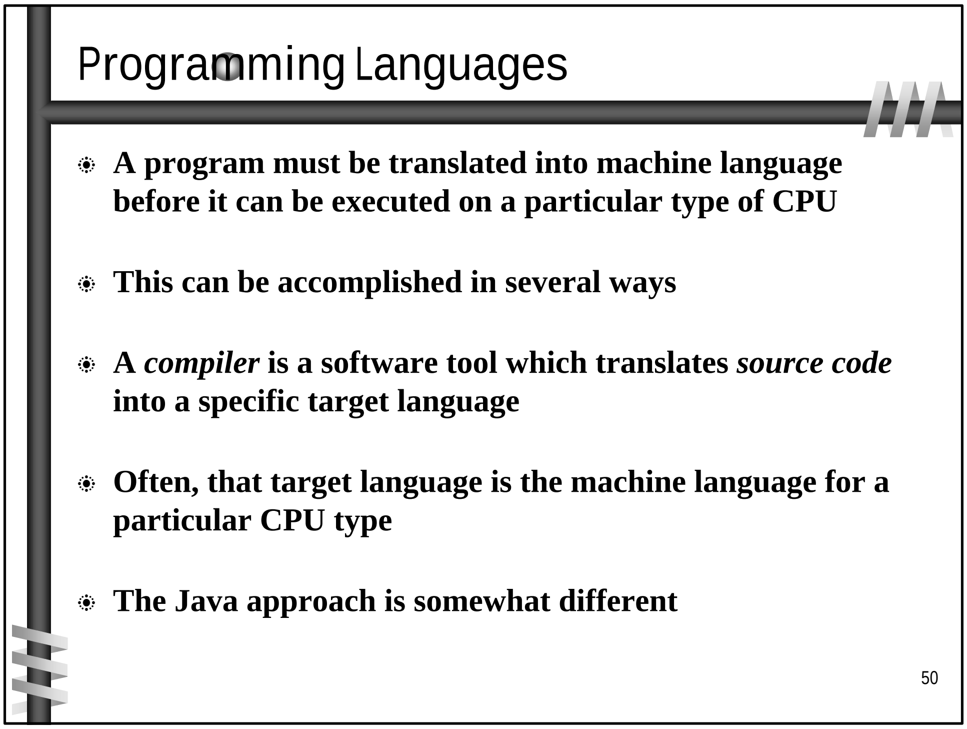


Programming Language Levels

- There are four programming language levels:
  - machine language
  - assembly language
  - high-level language
  - fourth-generation language

- Each type of CPU has its own specific *machine language*

- The other levels were created to make it easier for a human being to write programs



Programming Languages

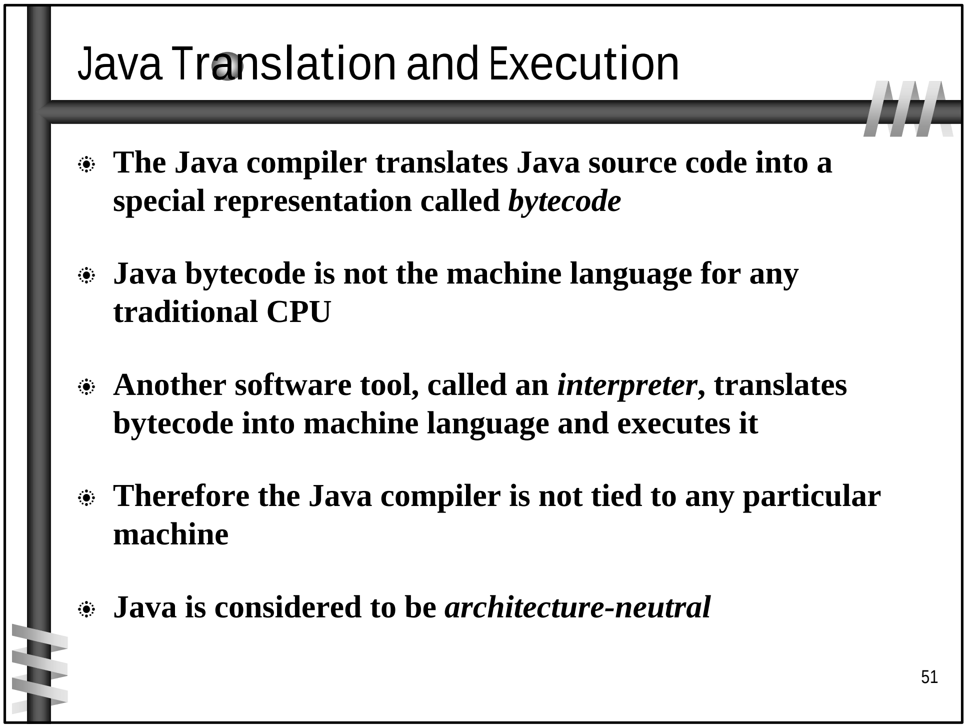
- A program must be translated into machine language before it can be executed on a particular type of CPU

- This can be accomplished in several ways

- A *compiler* is a software tool which translates *source code* into a specific target language

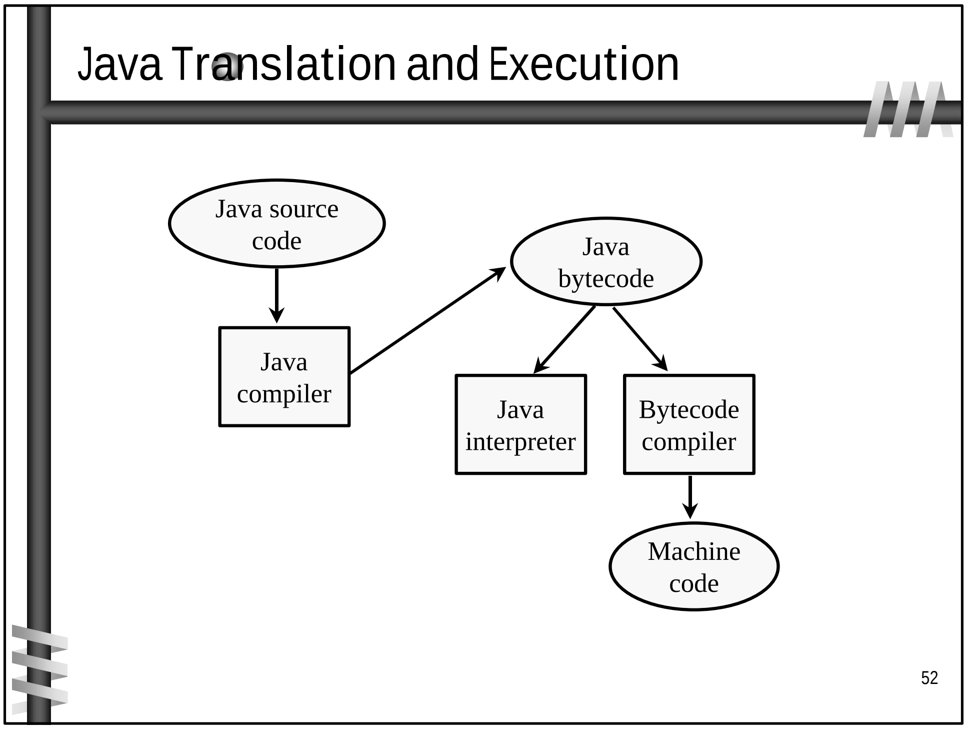
- Often, that target language is the machine language for a particular CPU type

- The Java approach is somewhat different



Java Translation and Execution

- **The Java compiler translates Java source code into a special representation called *bytecode***
- **Java bytecode is not the machine language for any traditional CPU**
- **Another software tool, called an *interpreter*, translates bytecode into machine language and executes it**
- **Therefore the Java compiler is not tied to any particular machine**
- **Java is considered to be *architecture-neutral***



Java Translation and Execution

Development Environments

- There are many development environments which develop Java software:
  - Sun Java Software Development Kit (SDK)
  - Borland JBuilder
  - MetroWork CodeWarrior
  - Microsoft Visual J++
  - Symantec Café

- Though the details of these environments differ, the basic compilation and execution process is essentially the same

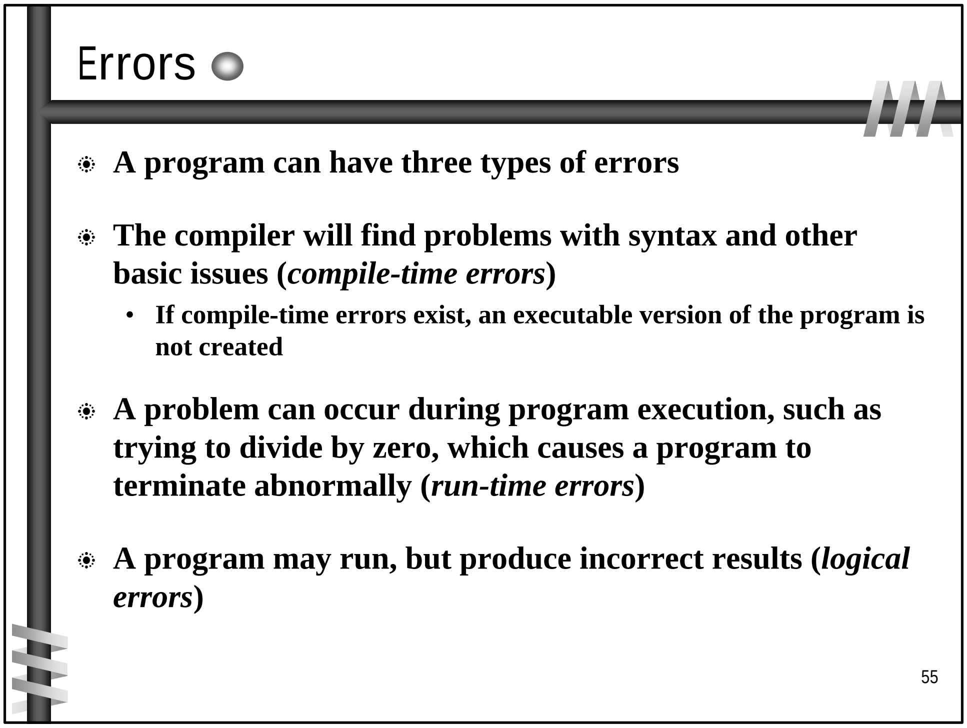
Syntax and Semantics

- The *syntax rules* of a language define how we can put symbols, reserved words, and identifiers together to make a valid program

- The *semantics* of a program statement define what that statement means (its purpose or role in a program)

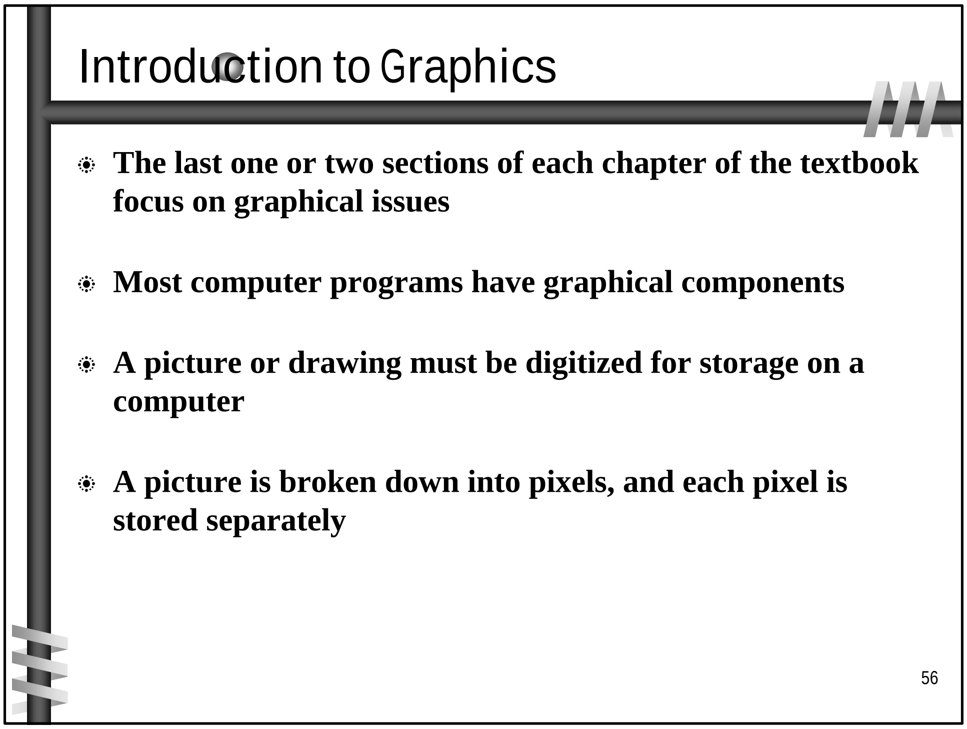
- A program that is syntactically correct is not necessarily logically (semantically) correct

- A program will always do what we tell it to do, not what we <u>meant</u> to tell it to do



Errors

- A program can have three types of errors
- The compiler will find problems with syntax and other basic issues (*compile-time errors*)
  - If compile-time errors exist, an executable version of the program is not created
- A problem can occur during program execution, such as trying to divide by zero, which causes a program to terminate abnormally (*run-time errors*)
- A program may run, but produce incorrect results (*logical errors*)



Introduction to Graphics

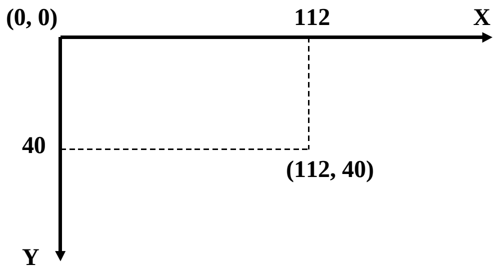
- The last one or two sections of each chapter of the textbook focus on graphical issues
- Most computer programs have graphical components
- A picture or drawing must be digitized for storage on a computer
- A picture is broken down into pixels, and each pixel is stored separately

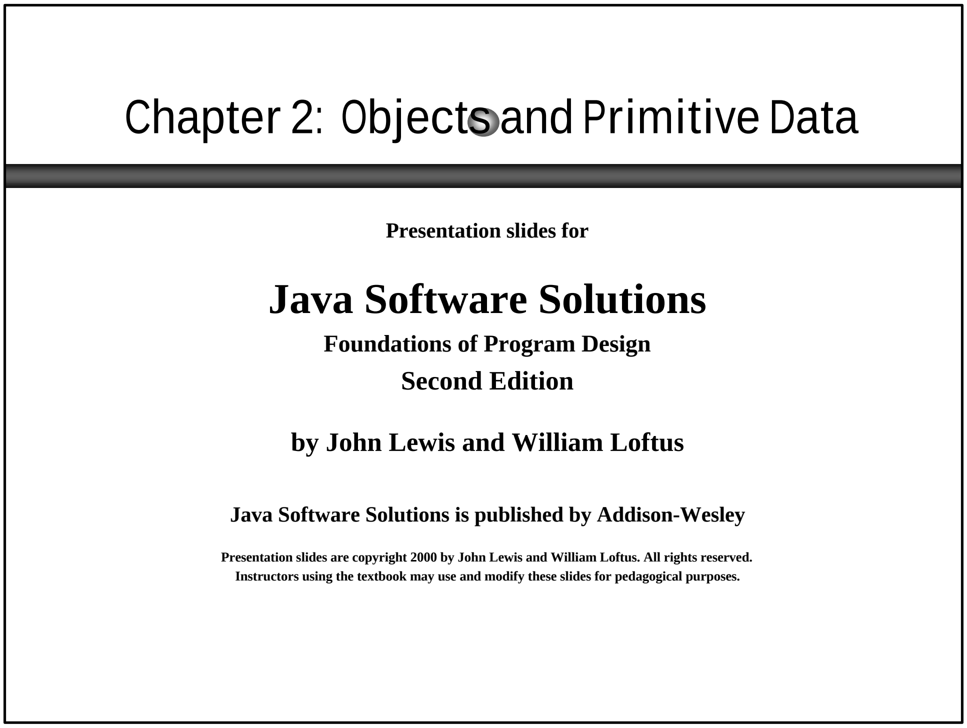
Representing Color

- **A black and white picture can be stored using one bit per pixel (0 = white and 1 = black)**
- **A color picture requires more information, and there are several techniques for representing a particular color**
- **For example, every color can be represented as a mixture of the three primary colors Red, Green, and Blue**
- **In Java, each color is represented by three numbers between 0 and 255 that are collectively called an *RGB value***

Coordinate Systems

- **Each pixel can be identified using a two-dimensional coordinate system**
- **When referring to a pixel in a Java program, we use a coordinate system with the origin in the upper left corner**





Chapter 2: Objects and Primitive Data

Presentation slides for

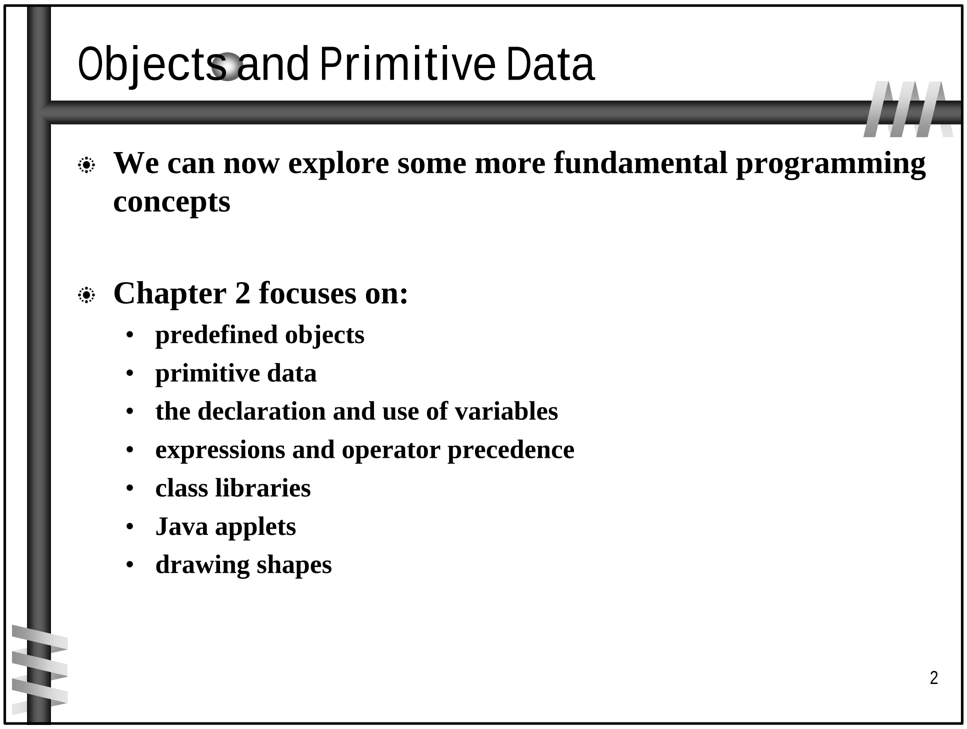
Java Software Solutions

Foundations of Program Design

Second Edition

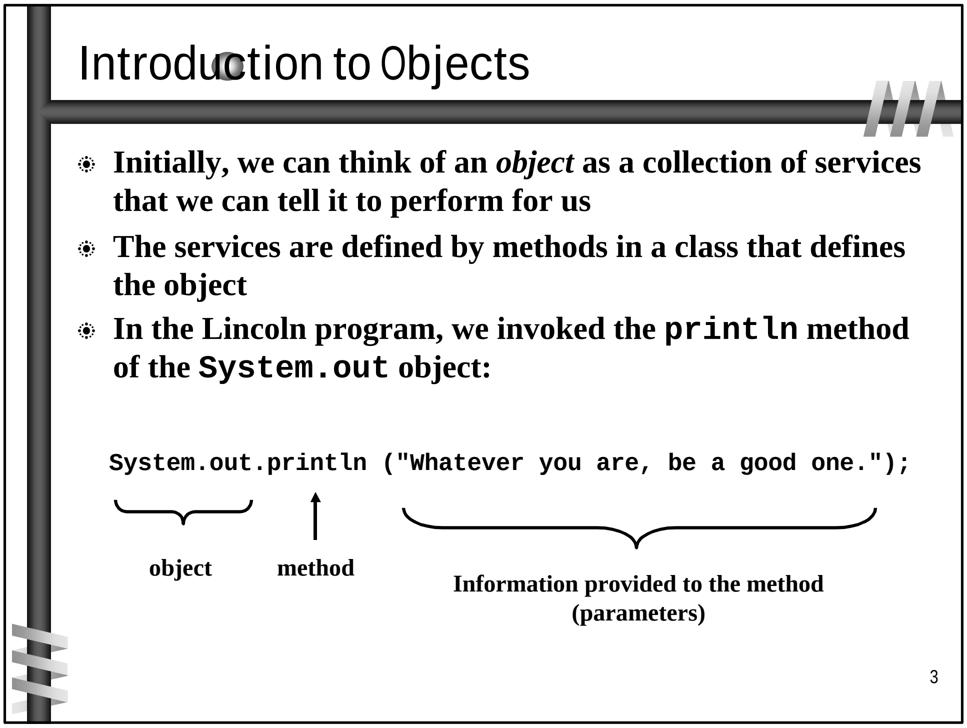
by John Lewis and William Loftus

Java Software Solutions is published by Addison-Wesley



Objects and Primitive Data

- **We can now explore some more fundamental programming concepts**
- **Chapter 2 focuses on:**
  - **predefined objects**
  - **primitive data**
  - **the declaration and use of variables**
  - **expressions and operator precedence**
  - **class libraries**
  - **Java applets**
  - **drawing shapes**



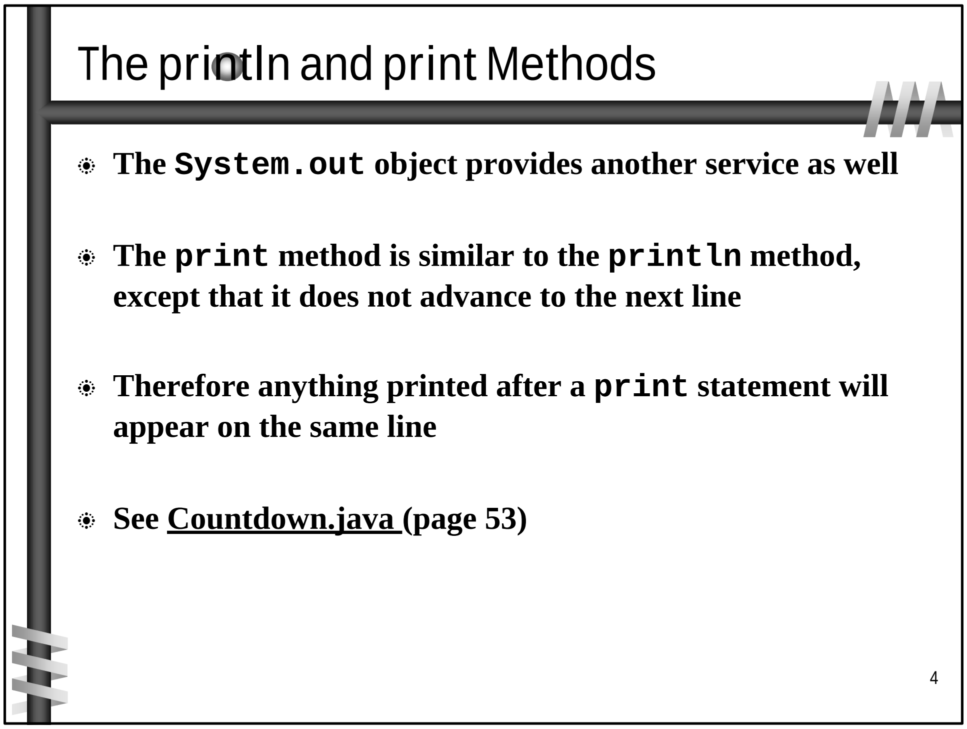
Introduction to Objects

- **Initially, we can think of an *object* as a collection of services that we can tell it to perform for us**
- **The services are defined by methods in a class that defines the object**
- **In the Lincoln program, we invoked the `println` method of the `System.out` object:**

```
System.out.println ("Whatever you are, be a good one.");
```

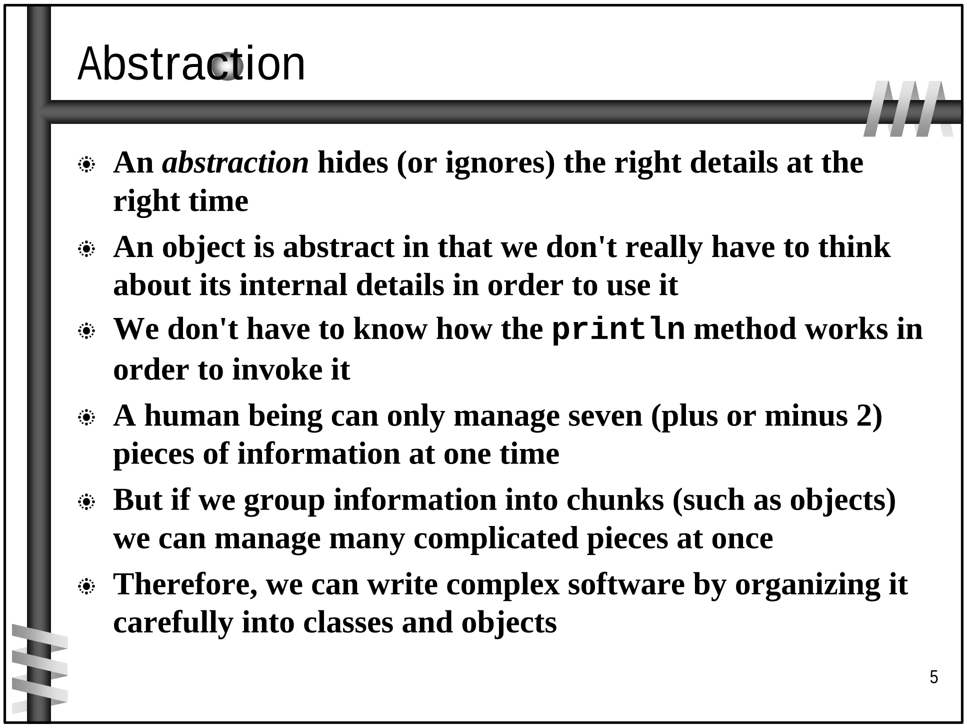
object **method**

Information provided to the method (parameters)



The println and print Methods

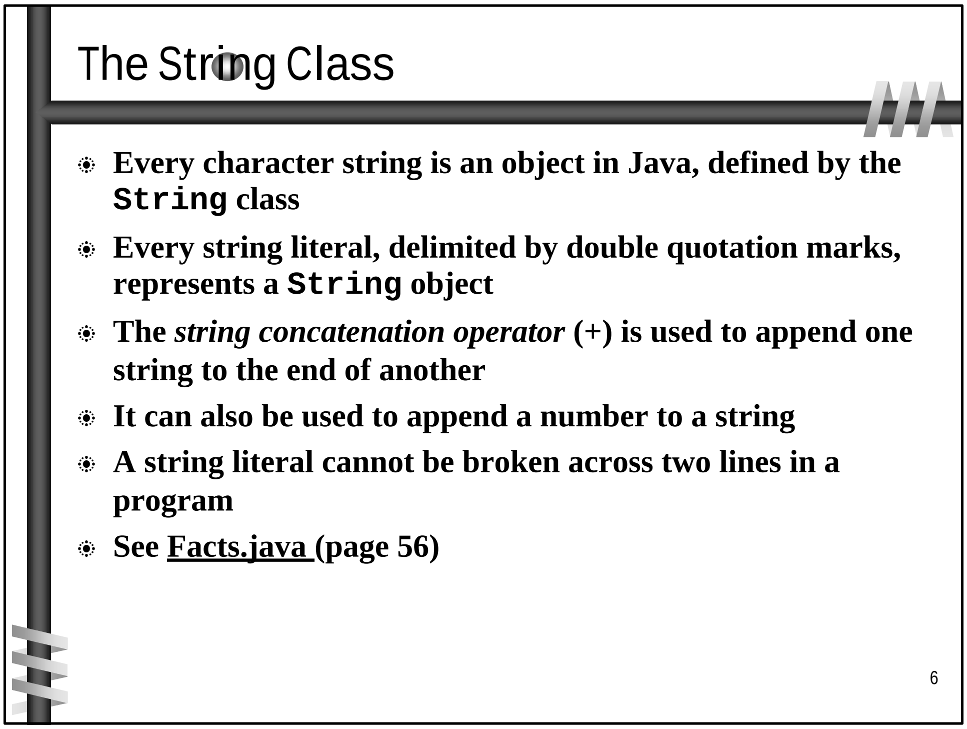
- **The `System.out` object provides another service as well**
- **The `print` method is similar to the `println` method, except that it does not advance to the next line**
- **Therefore anything printed after a `print` statement will appear on the same line**
- **See Countdown.java (page 53)**



Abstraction

- An *abstraction* hides (or ignores) the right details at the right time
- An object is abstract in that we don't really have to think about its internal details in order to use it
- We don't have to know how the `println` method works in order to invoke it
- A human being can only manage seven (plus or minus 2) pieces of information at one time
- But if we group information into chunks (such as objects) we can manage many complicated pieces at once
- Therefore, we can write complex software by organizing it carefully into classes and objects

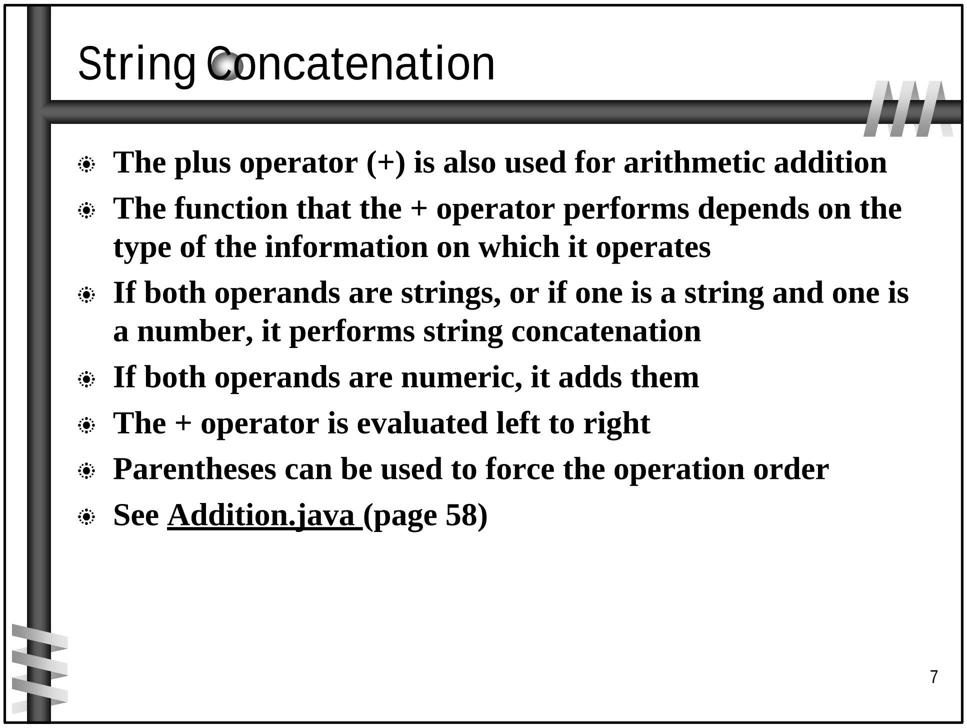
5



The String Class

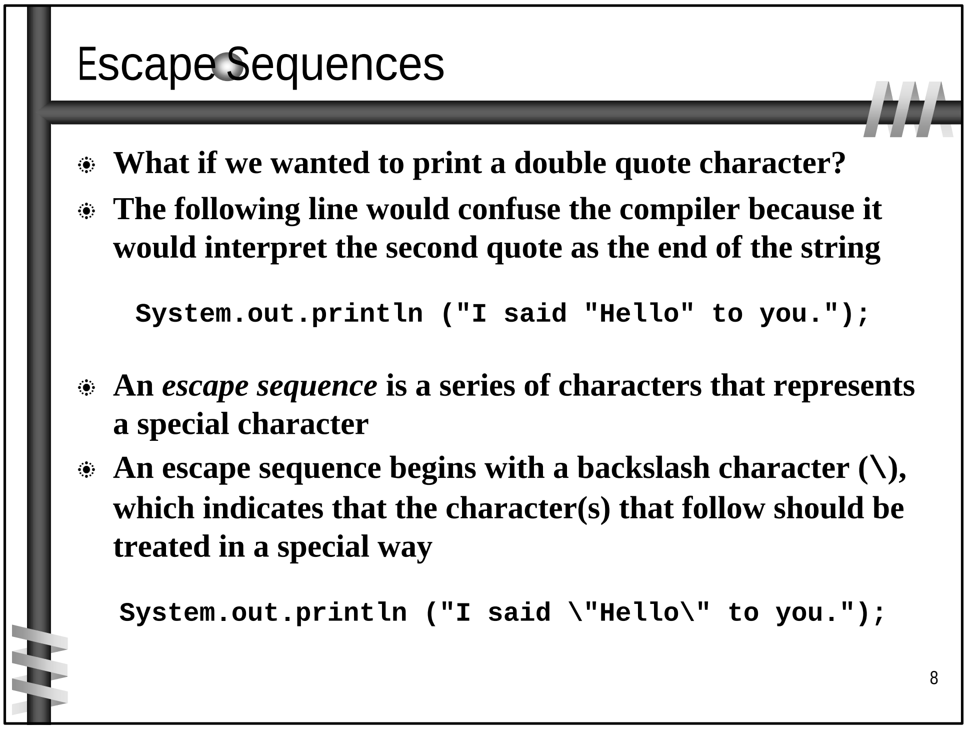
- Every character string is an object in Java, defined by the `String` class
- Every string literal, delimited by double quotation marks, represents a `String` object
- The *string concatenation operator* (+) is used to append one string to the end of another
- It can also be used to append a number to a string
- A string literal cannot be broken across two lines in a program
- See Facts.java (page 56)

6



String Concatenation

- The plus operator (+) is also used for arithmetic addition
- The function that the + operator performs depends on the type of the information on which it operates
- If both operands are strings, or if one is a string and one is a number, it performs string concatenation
- If both operands are numeric, it adds them
- The + operator is evaluated left to right
- Parentheses can be used to force the operation order
- See Addition.java (page 58)



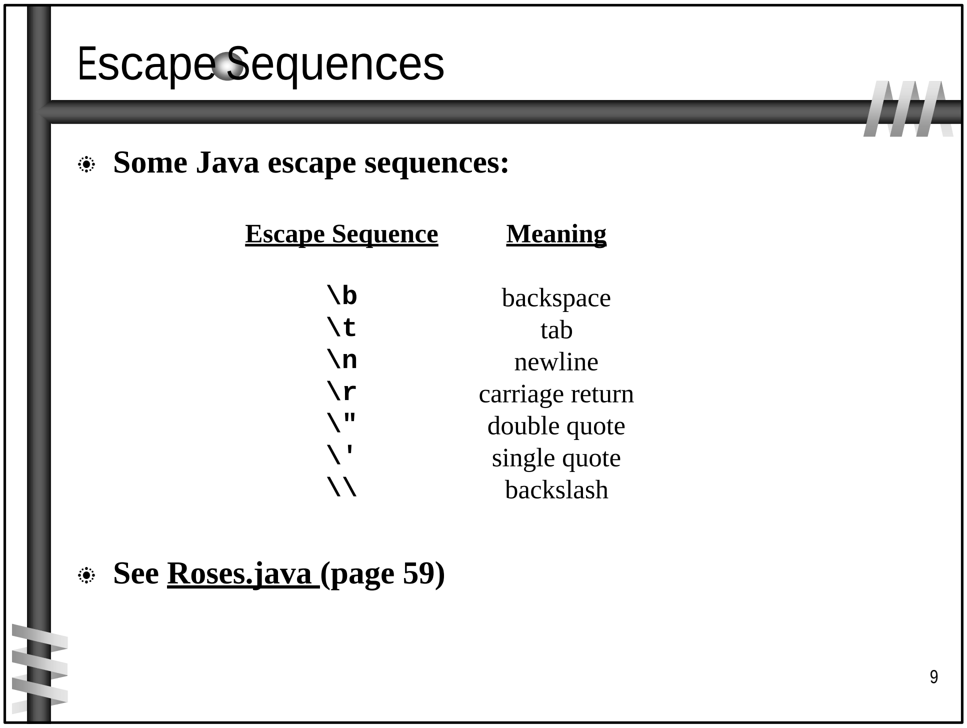
Escape Sequences

- What if we wanted to print a double quote character?
- The following line would confuse the compiler because it would interpret the second quote as the end of the string

```
System.out.println ("I said "Hello" to you.");
```

- An *escape sequence* is a series of characters that represents a special character
- An escape sequence begins with a backslash character (\), which indicates that the character(s) that follow should be treated in a special way

```
System.out.println ("I said \"Hello\" to you.");
```

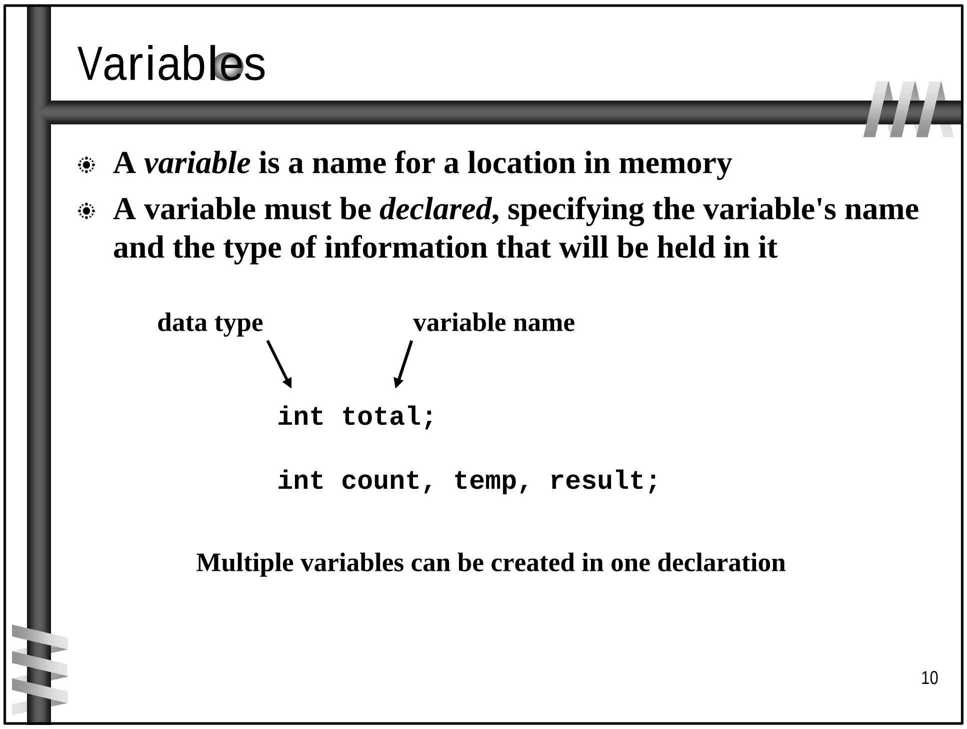


Escape Sequences

- Some Java escape sequences:

Escape Sequence	Meaning
`\b`	backspace
`\t`	tab
`\n`	newline
`\r`	carriage return
`\"`	double quote
`\'`	single quote
`\\`	backslash

- See Roses.java (page 59)



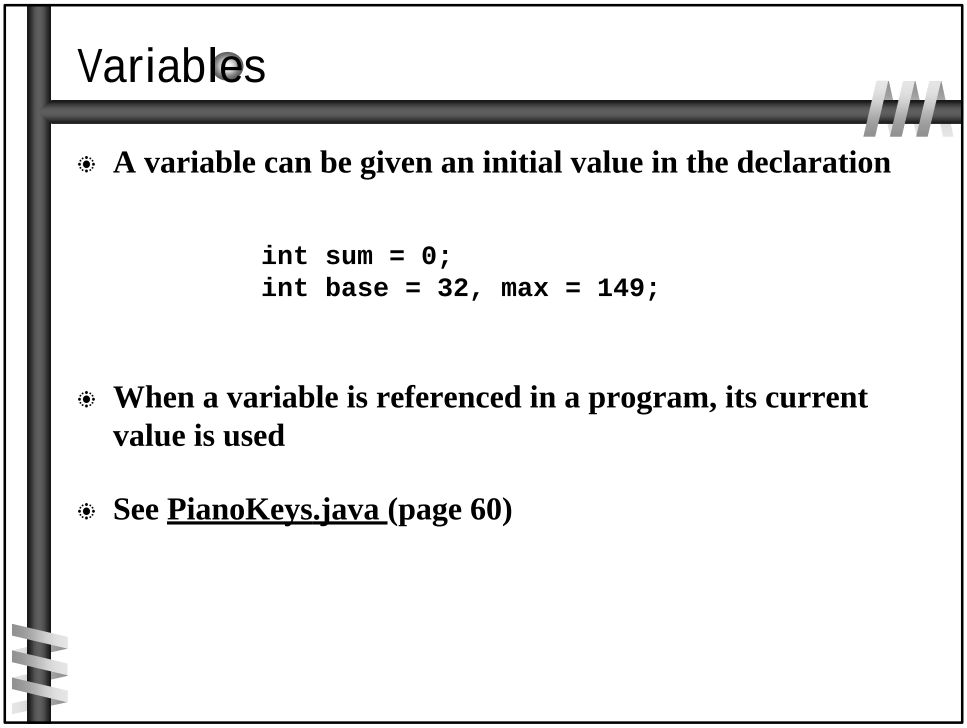
Variables

- **A *variable* is a name for a location in memory**
- **A variable must be *declared*, specifying the variable's name and the type of information that will be held in it**

data type **variable name**

```
int total;

int count, temp, result;
```

Multiple variables can be created in one declaration

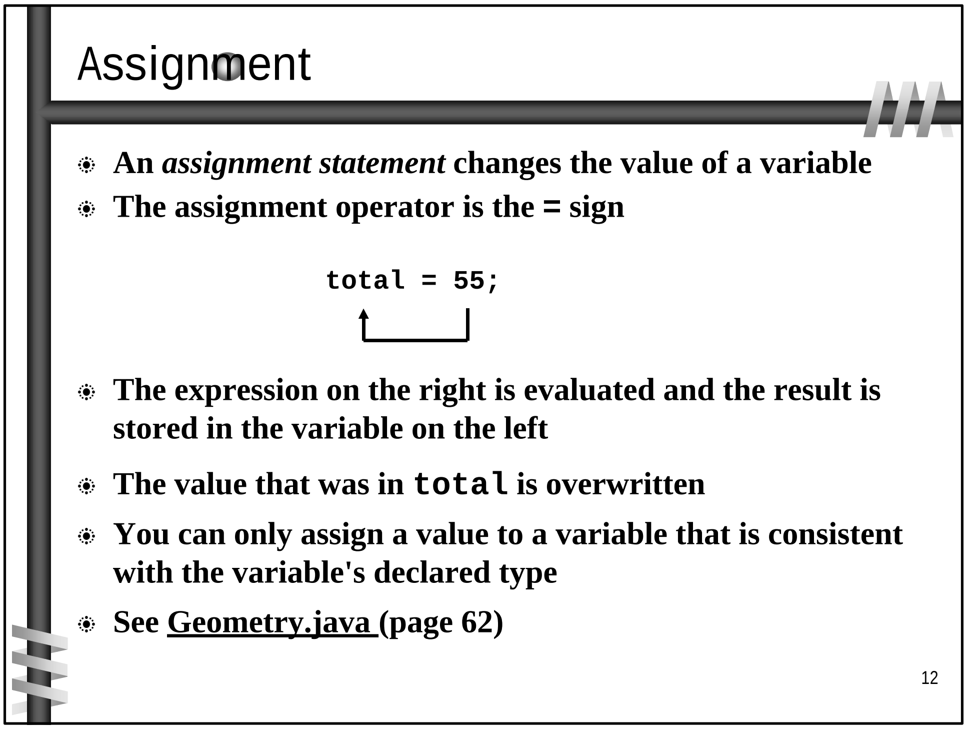


Variables

- A variable can be given an initial value in the declaration

```
int sum = 0;
int base = 32, max = 149;
```

- When a variable is referenced in a program, its current value is used
- See PianoKeys.java (page 60)

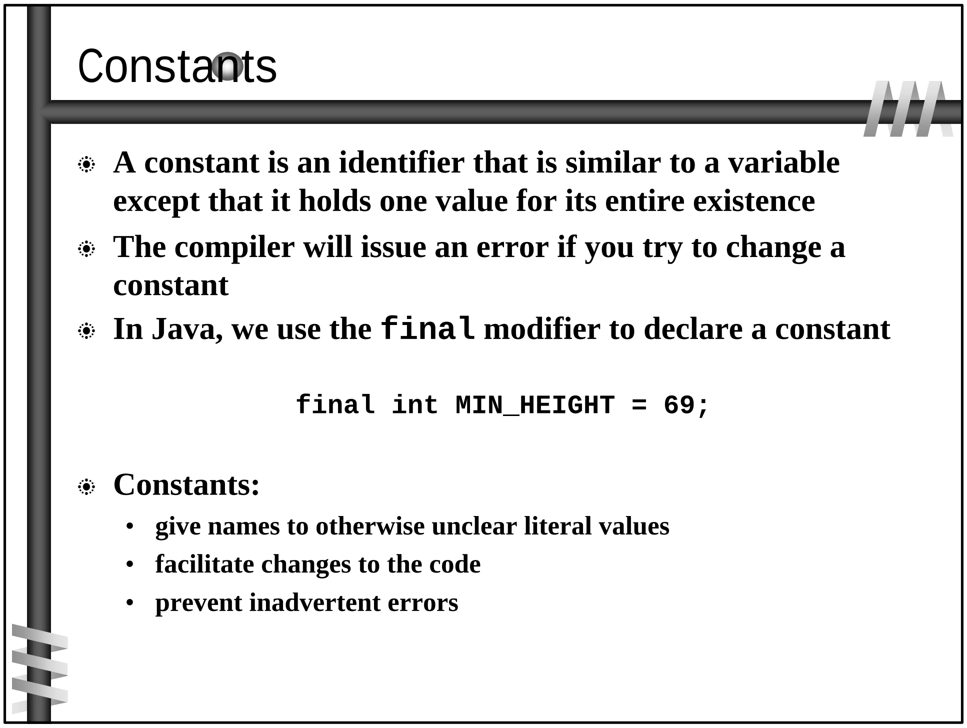


Assignment

- An *assignment statement* changes the value of a variable
- The assignment operator is the = sign

```
total = 55;
```

- The expression on the right is evaluated and the result is stored in the variable on the left
- The value that was in `total` is overwritten
- You can only assign a value to a variable that is consistent with the variable's declared type
- See Geometry.java (page 62)

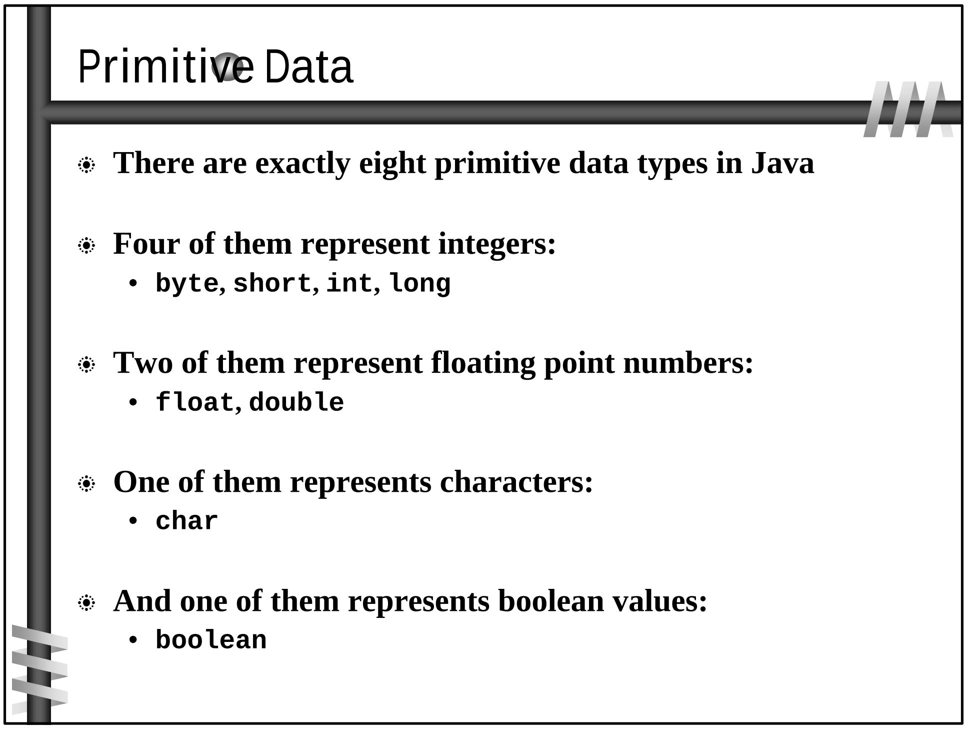


Constants

- A constant is an identifier that is similar to a variable except that it holds one value for its entire existence
- The compiler will issue an error if you try to change a constant
- In Java, we use the `final` modifier to declare a constant

```
final int MIN_HEIGHT = 69;
```

- Constants:
  - give names to otherwise unclear literal values
  - facilitate changes to the code
  - prevent inadvertent errors



Primitive Data

- There are exactly eight primitive data types in Java
- Four of them represent integers:
  - `byte`, `short`, `int`, `long`
- Two of them represent floating point numbers:
  - `float`, `double`
- One of them represents characters:
  - `char`
- And one of them represents boolean values:
  - `boolean`

Numeric Primitive Data

- **The difference between the various numeric primitive types is their size, and therefore the values they can store:**

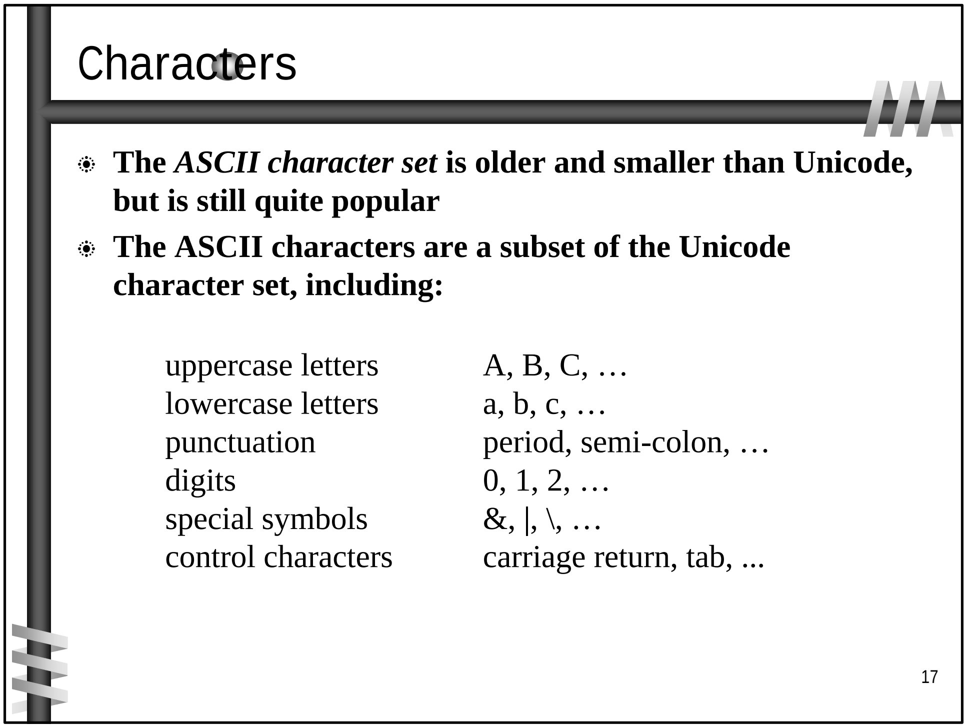
Type	**Storage**	**Min Value**	**Max Value**
byte	**8 bits**	**-128**	**127**
short	**16 bits**	**-32,768**	**32,767**
int	**32 bits**	**-2,147,483,648**	**2,147,483,647**
long	**64 bits**	**< -9 x 10^{18}**	**> 9 x 10^{18}**
float	**32 bits**	**+/- 3.4 x 10^{38} with 7 significant digits**	
double	**64 bits**	**+/- 1.7 x 10^{308} with 15 significant digits**	

Characters

- **A `char` variable stores a single character from the *Unicode character set***
- **A *character set* is an ordered list of characters, and each character corresponds to a unique number**
- **The Unicode character set uses sixteen bits per character, allowing for 65,536 unique characters**
- **It is an international character set, containing symbols and characters from many world languages**
- **Character literals are delimited by single quotes:**

```
'a'   'X'    '7'    '$'    ','    '\n'
```

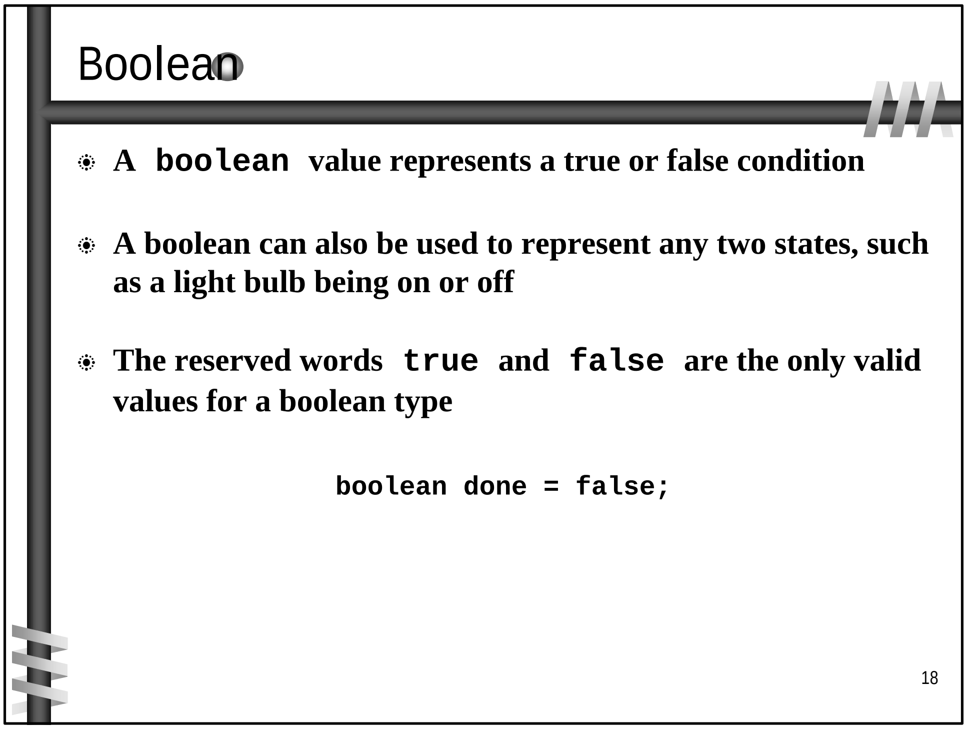
16



Characters

- The *ASCII character set* is older and smaller than Unicode, but is still quite popular
- The ASCII characters are a subset of the Unicode character set, including:

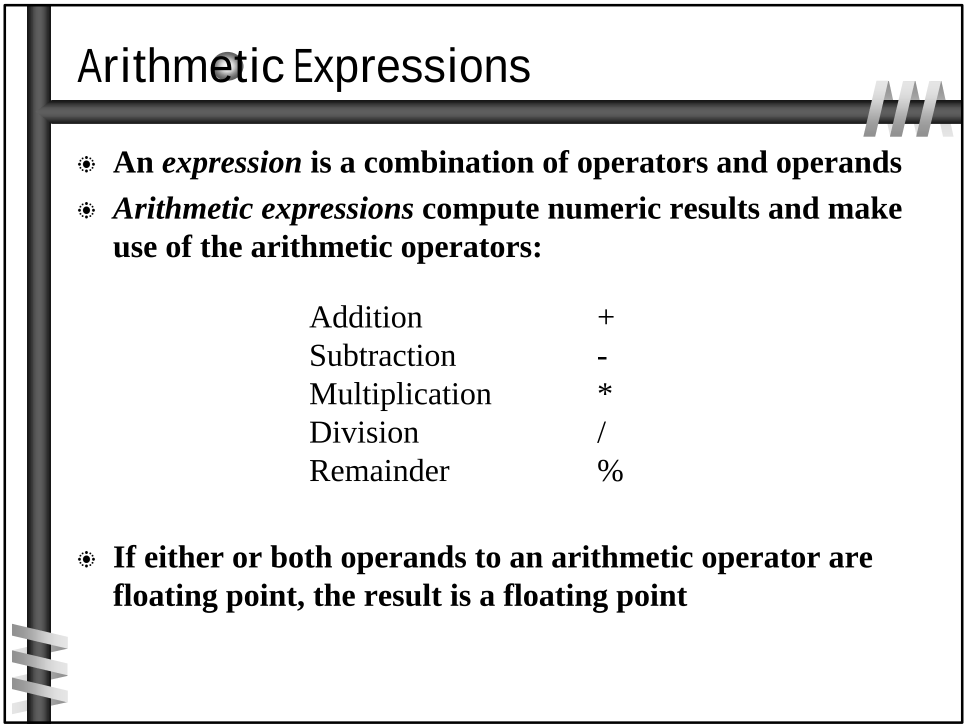
uppercase letters	A, B, C, …
lowercase letters	a, b, c, …
punctuation	period, semi-colon, …
digits	0, 1, 2, …
special symbols	&, |, \, …
control characters	carriage return, tab, ...



Boolean

- A `boolean` value represents a true or false condition
- A boolean can also be used to represent any two states, such as a light bulb being on or off
- The reserved words `true` and `false` are the only valid values for a boolean type

```
boolean done = false;
```

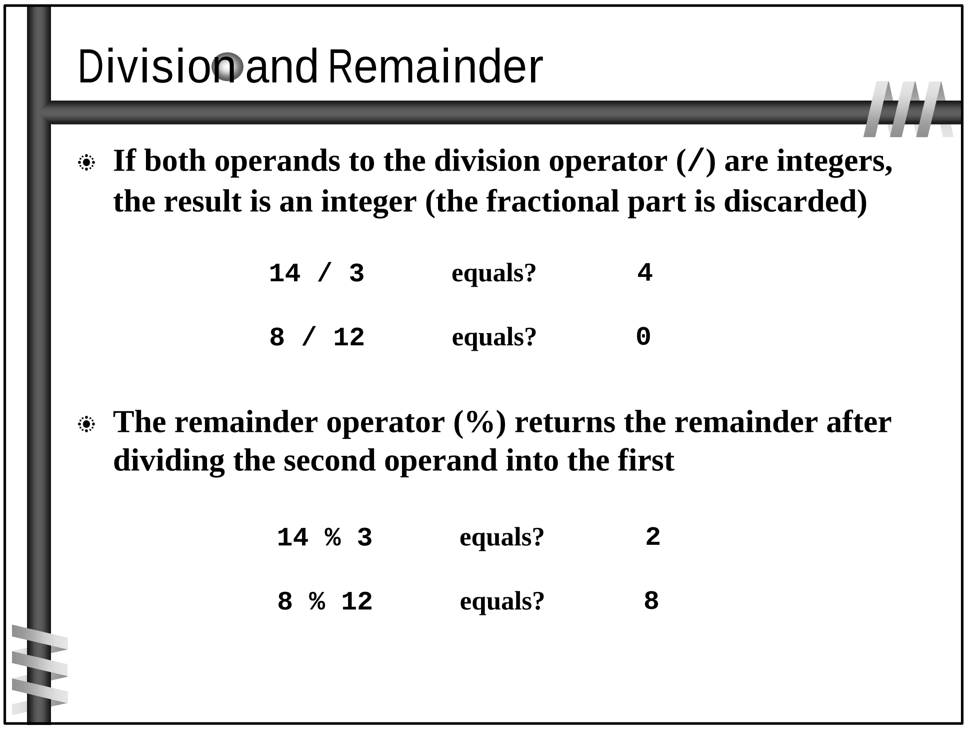


Arithmetic Expressions

- An *expression* is a combination of operators and operands
- *Arithmetic expressions* compute numeric results and make use of the arithmetic operators:

Addition	+
Subtraction	-
Multiplication	*
Division	/
Remainder	%

- If either or both operands to an arithmetic operator are floating point, the result is a floating point



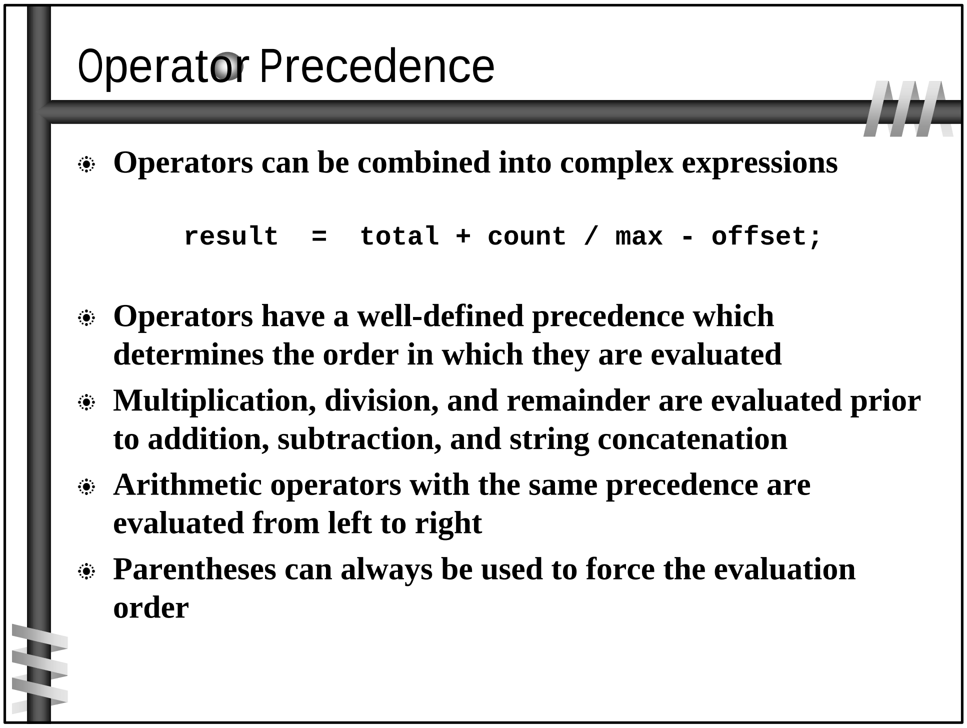
Division and Remainder

- If both operands to the division operator (`/`) are integers, the result is an integer (the fractional part is discarded)

`14 / 3`	equals?	`4`
`8 / 12`	equals?	`0`

- The remainder operator (%) returns the remainder after dividing the second operand into the first

`14 % 3`	equals?	`2`
`8 % 12`	equals?	`8`

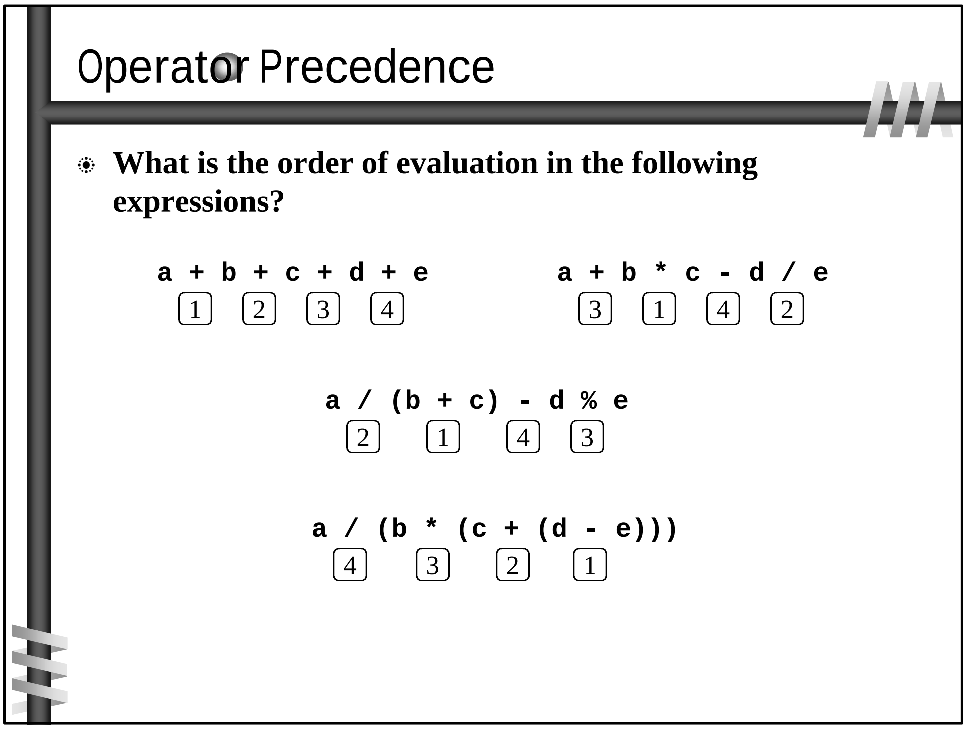


Operator Precedence

- Operators can be combined into complex expressions

```
result  =  total + count / max - offset;
```

- Operators have a well-defined precedence which determines the order in which they are evaluated
- Multiplication, division, and remainder are evaluated prior to addition, subtraction, and string concatenation
- Arithmetic operators with the same precedence are evaluated from left to right
- Parentheses can always be used to force the evaluation order



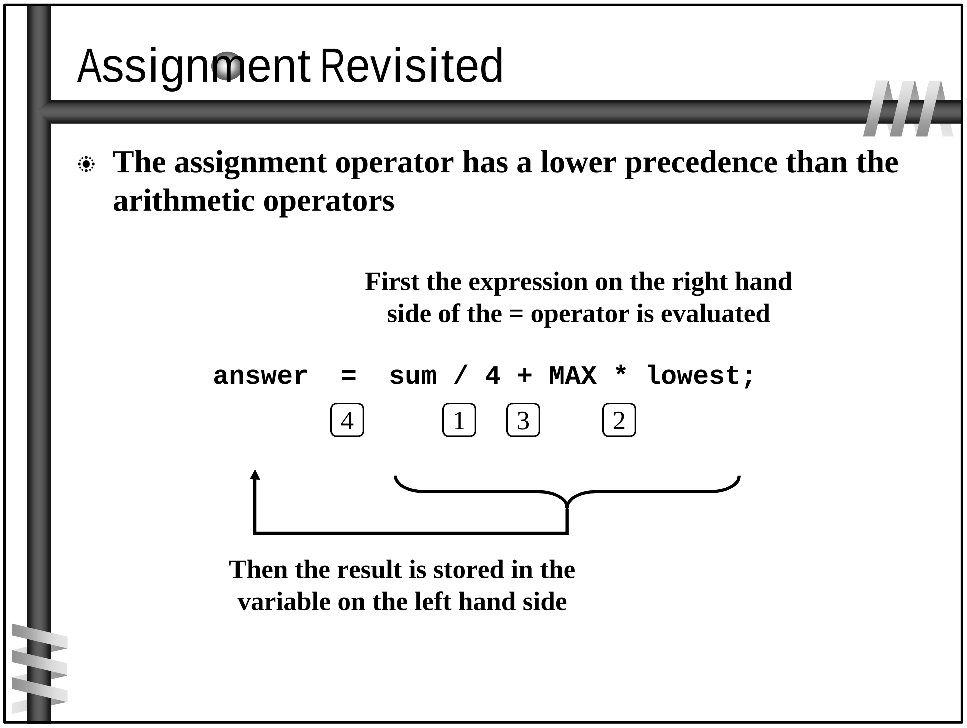
Operator Precedence

- What is the order of evaluation in the following expressions?

```
a + b + c + d + e          a + b * c - d / e
  1   2   3   4              3   1   4   2
```

```
a / (b + c) - d % e
  2    1    4   3
```

```
a / (b * (c + (d - e)))
  4    3    2    1
```



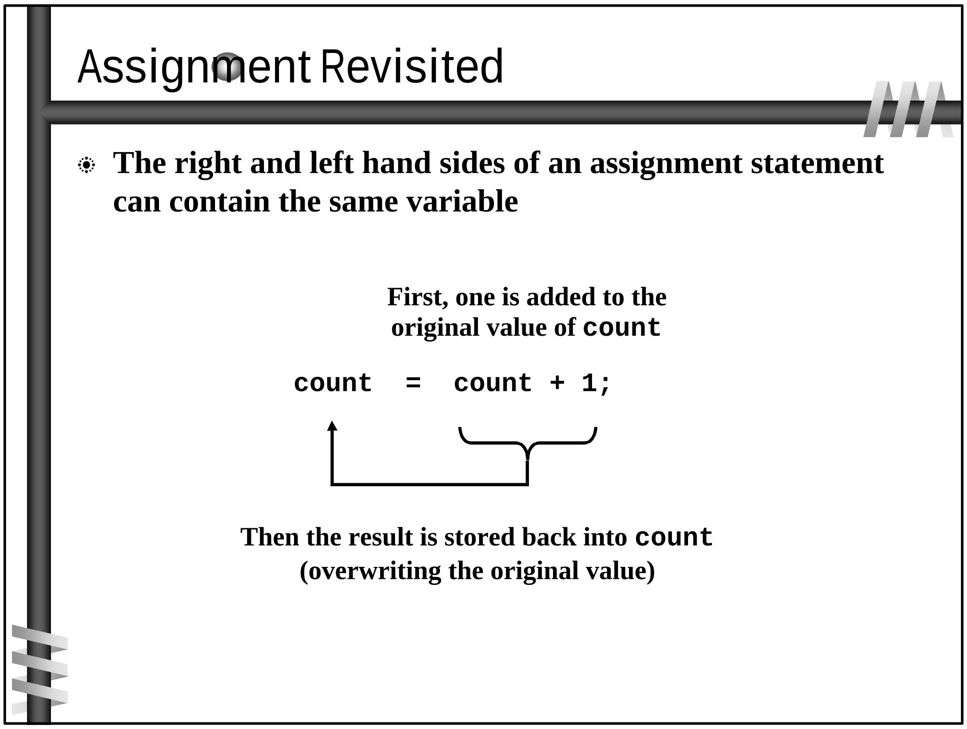
Assignment Revisited

- **The assignment operator has a lower precedence than the arithmetic operators**

First the expression on the right hand side of the = operator is evaluated

```
answer  =  sum / 4 + MAX * lowest;
        4      1   3     2
```

Then the result is stored in the variable on the left hand side



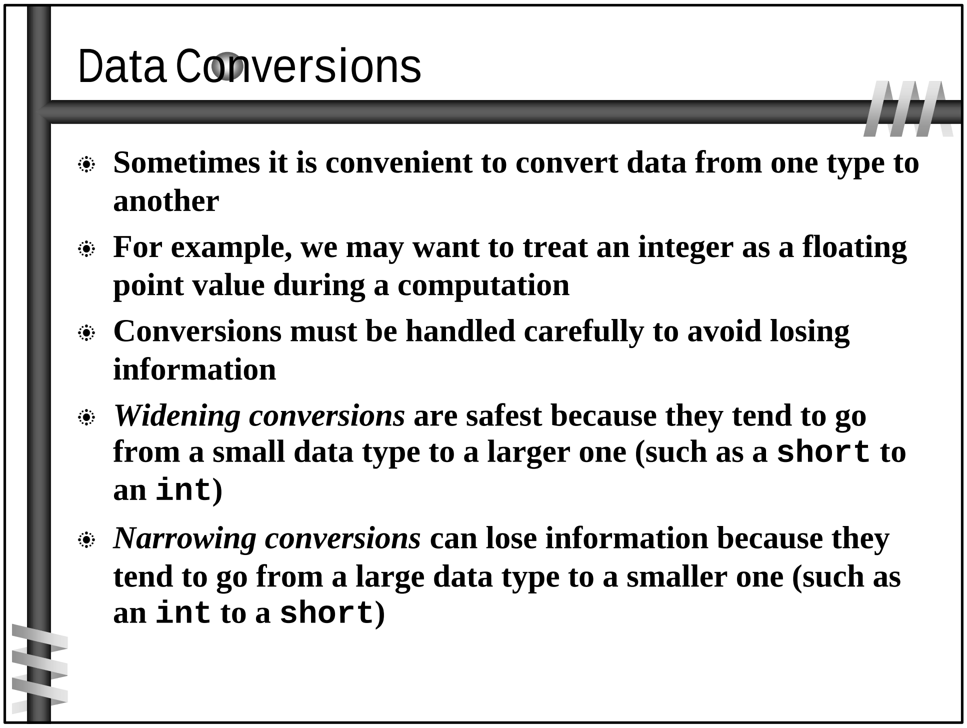
Assignment Revisited

- **The right and left hand sides of an assignment statement can contain the same variable**

First, one is added to the original value of `count`

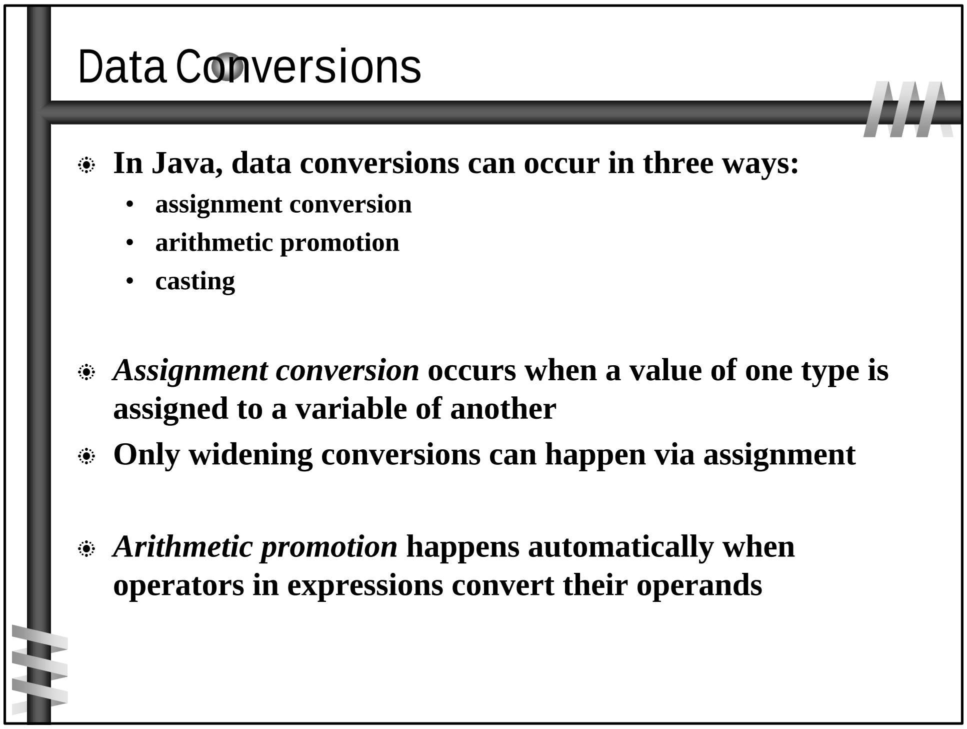
```
count  =  count + 1;
```

Then the result is stored back into `count` (overwriting the original value)



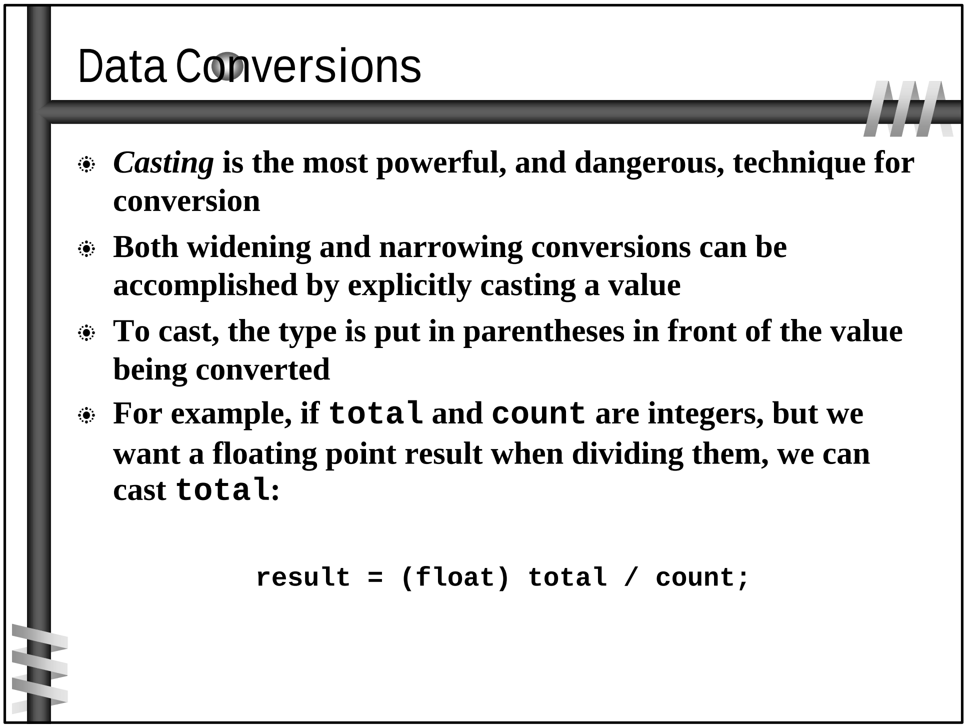
Data Conversions

- Sometimes it is convenient to convert data from one type to another
- For example, we may want to treat an integer as a floating point value during a computation
- Conversions must be handled carefully to avoid losing information
- *Widening conversions* are safest because they tend to go from a small data type to a larger one (such as a `short` to an `int`)
- *Narrowing conversions* can lose information because they tend to go from a large data type to a smaller one (such as an `int` to a `short`)



Data Conversions

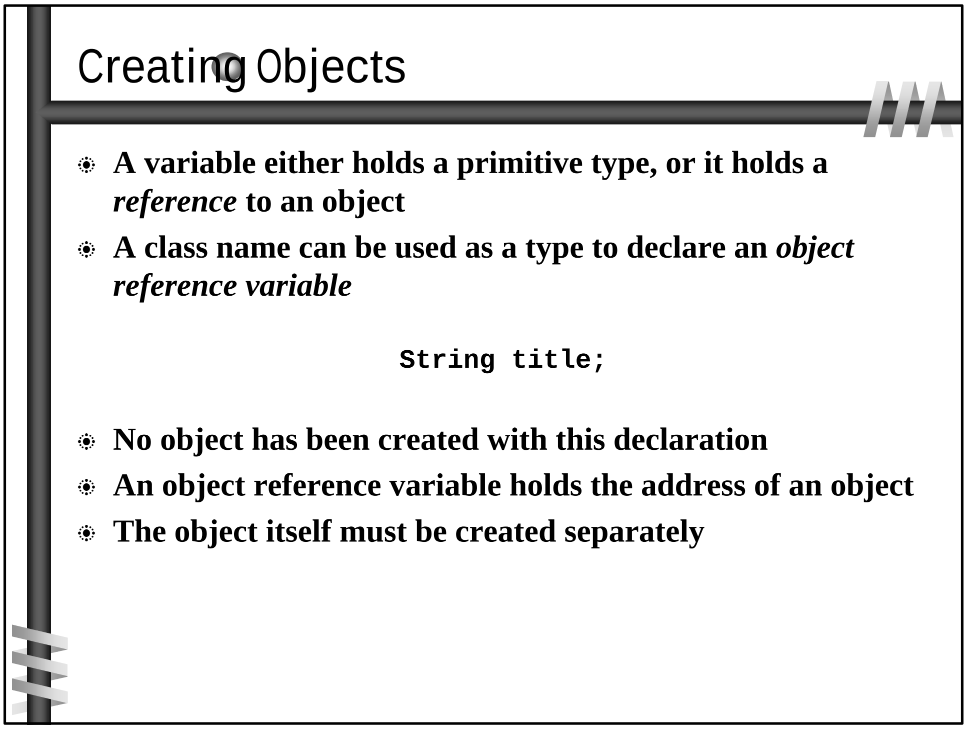
- In Java, data conversions can occur in three ways:
  - assignment conversion
  - arithmetic promotion
  - casting
- *Assignment conversion* occurs when a value of one type is assigned to a variable of another
- Only widening conversions can happen via assignment
- *Arithmetic promotion* happens automatically when operators in expressions convert their operands



Data Conversions

- *Casting* is the most powerful, and dangerous, technique for conversion
- Both widening and narrowing conversions can be accomplished by explicitly casting a value
- To cast, the type is put in parentheses in front of the value being converted
- For example, if `total` and `count` are integers, but we want a floating point result when dividing them, we can cast `total`:

```
result = (float) total / count;
```

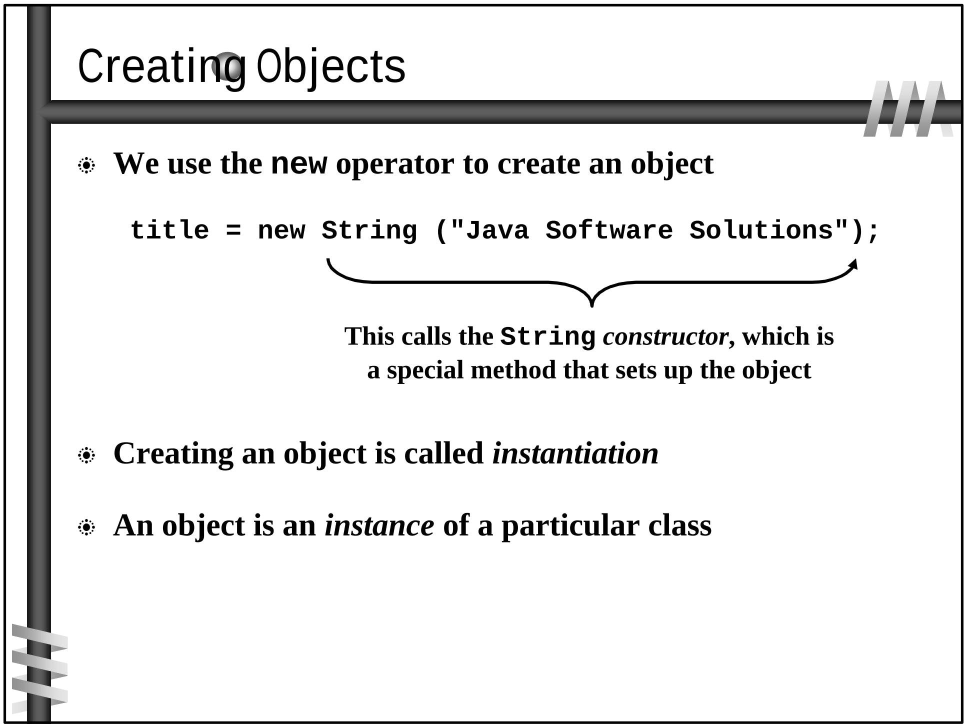


Creating Objects

- A variable either holds a primitive type, or it holds a *reference* to an object
- A class name can be used as a type to declare an *object reference variable*

```
String title;
```

- No object has been created with this declaration
- An object reference variable holds the address of an object
- The object itself must be created separately



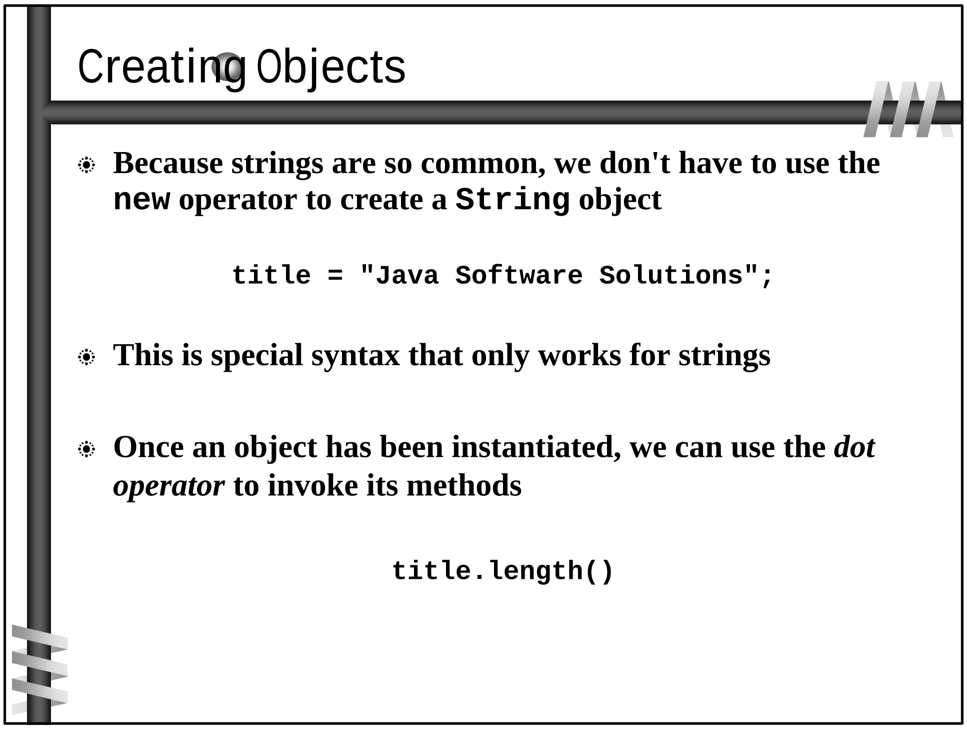
Creating Objects

- We use the `new` operator to create an object

```
title = new String ("Java Software Solutions");
```

This calls the `String` *constructor*, which is a special method that sets up the object

- Creating an object is called *instantiation*
- An object is an *instance* of a particular class



Creating Objects

- Because strings are so common, we don't have to use the `new` operator to create a `String` object

```
title = "Java Software Solutions";
```

- This is special syntax that only works for strings
- Once an object has been instantiated, we can use the *dot operator* to invoke its methods

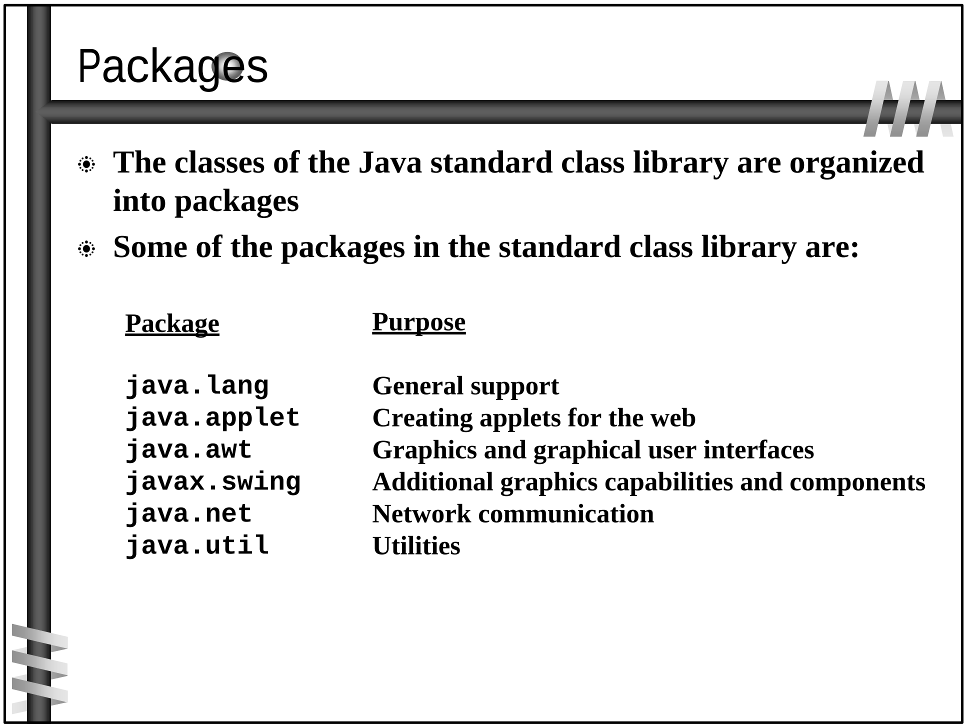
```
title.length()
```

String Methods

- **The `String` class has several methods that are useful for manipulating strings**
- **Many of the methods *return a value*, such as an integer or a new `String` object**
- **See the list of `String` methods on page 75 and in Appendix M**
- **See StringMutation.java (page 77)**

Class Libraries

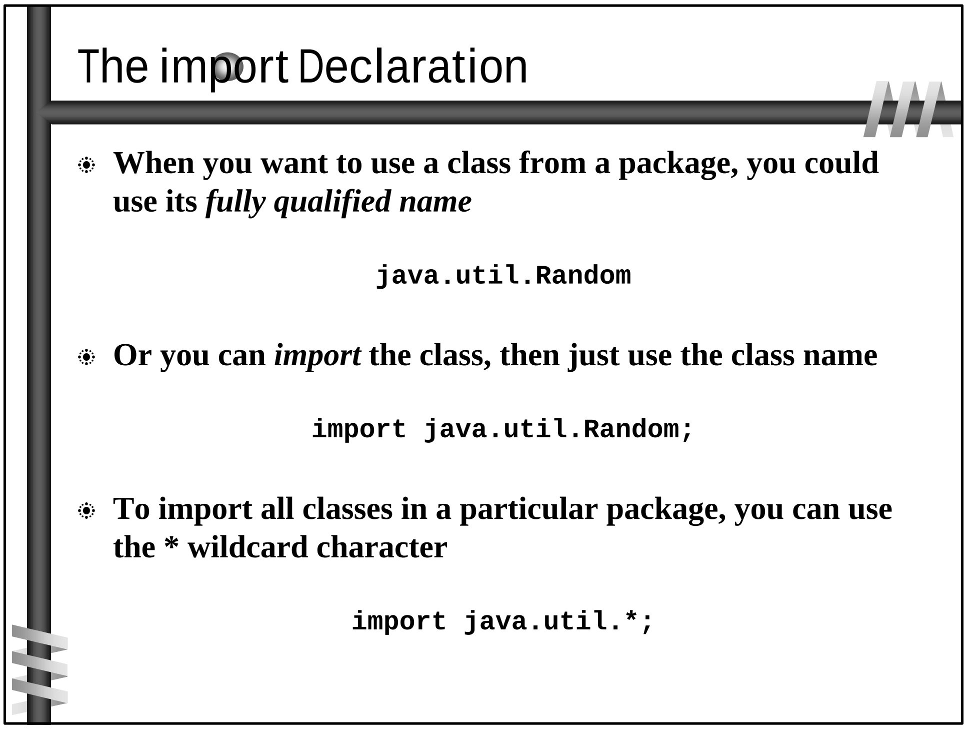
- **A *class library* is a collection of classes that we can use when developing programs**
- **There is a *Java standard class library* that is part of any Java development environment**
- **These classes are not part of the Java language per se, but we rely on them heavily**
- **The `System` class and the `String` class are part of the Java standard class library**
- **Other class libraries can be obtained through third party vendors, or you can create them yourself**



Packages

- The classes of the Java standard class library are organized into packages
- Some of the packages in the standard class library are:

Package	Purpose
`java.lang`	General support
`java.applet`	Creating applets for the web
`java.awt`	Graphics and graphical user interfaces
`javax.swing`	Additional graphics capabilities and components
`java.net`	Network communication
`java.util`	Utilities



The import Declaration

- When you want to use a class from a package, you could use its *fully qualified name*

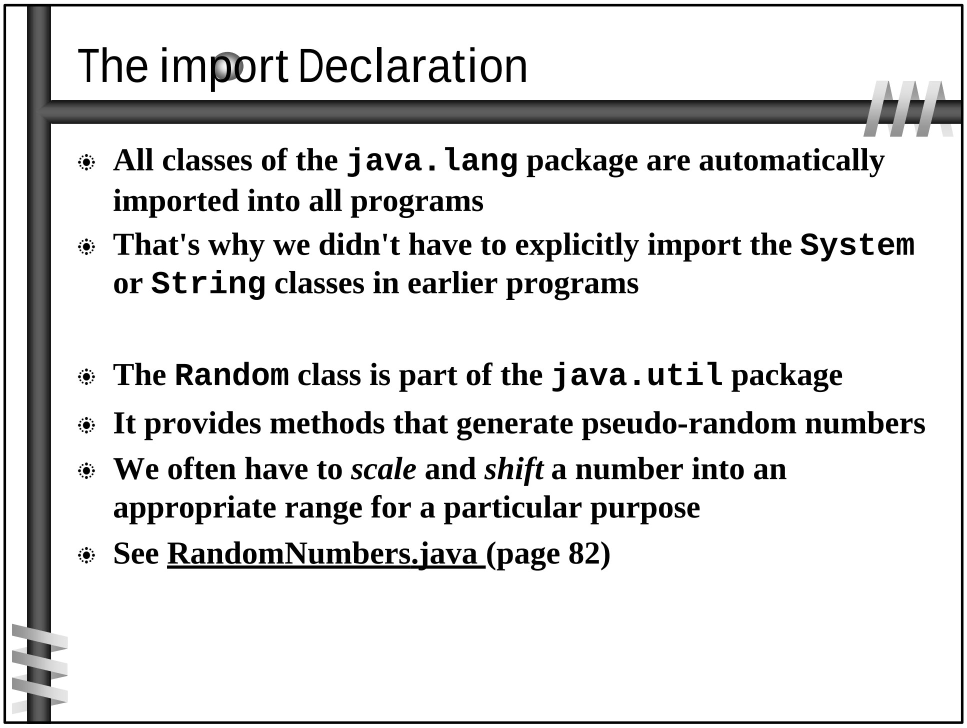
```
java.util.Random
```

- Or you can *import* the class, then just use the class name

```
import java.util.Random;
```

- To import all classes in a particular package, you can use the * wildcard character

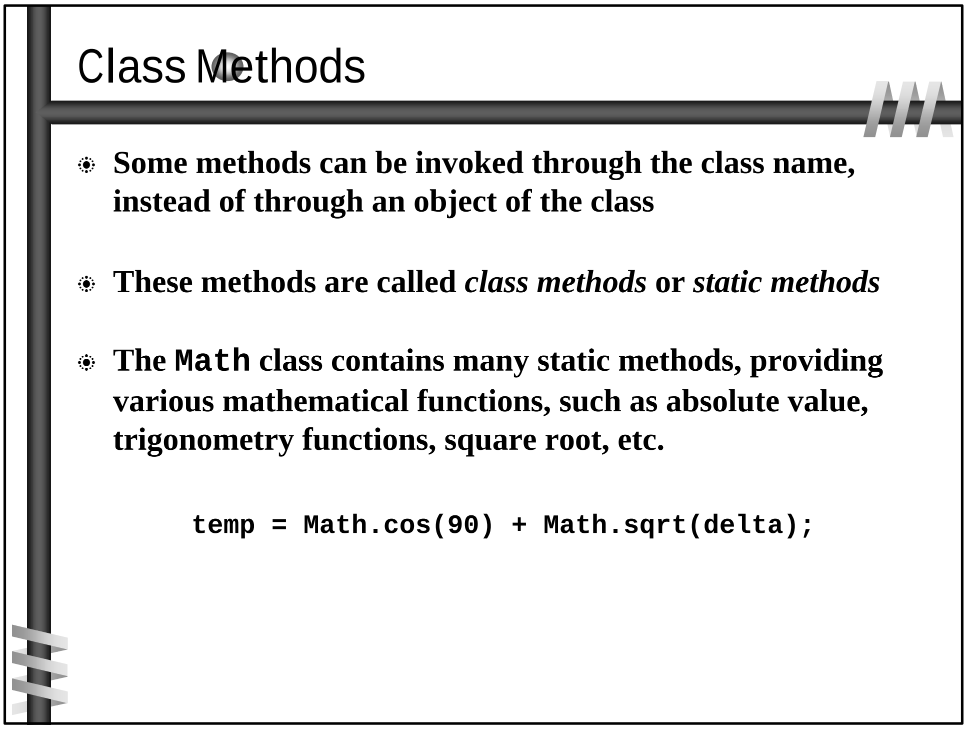
```
import java.util.*;
```



The import Declaration

- All classes of the `java.lang` package are automatically imported into all programs
- That's why we didn't have to explicitly import the `System` or `String` classes in earlier programs

- The `Random` class is part of the `java.util` package
- It provides methods that generate pseudo-random numbers
- We often have to *scale* and *shift* a number into an appropriate range for a particular purpose
- See RandomNumbers.java (page 82)



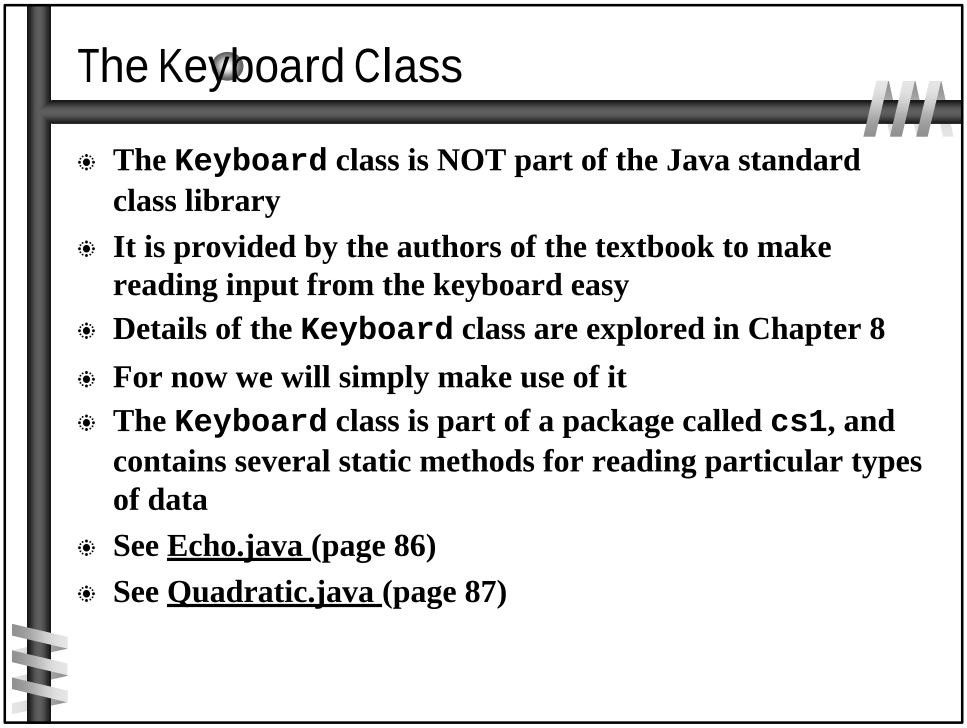
Class Methods

- Some methods can be invoked through the class name, instead of through an object of the class

- These methods are called *class methods* or *static methods*

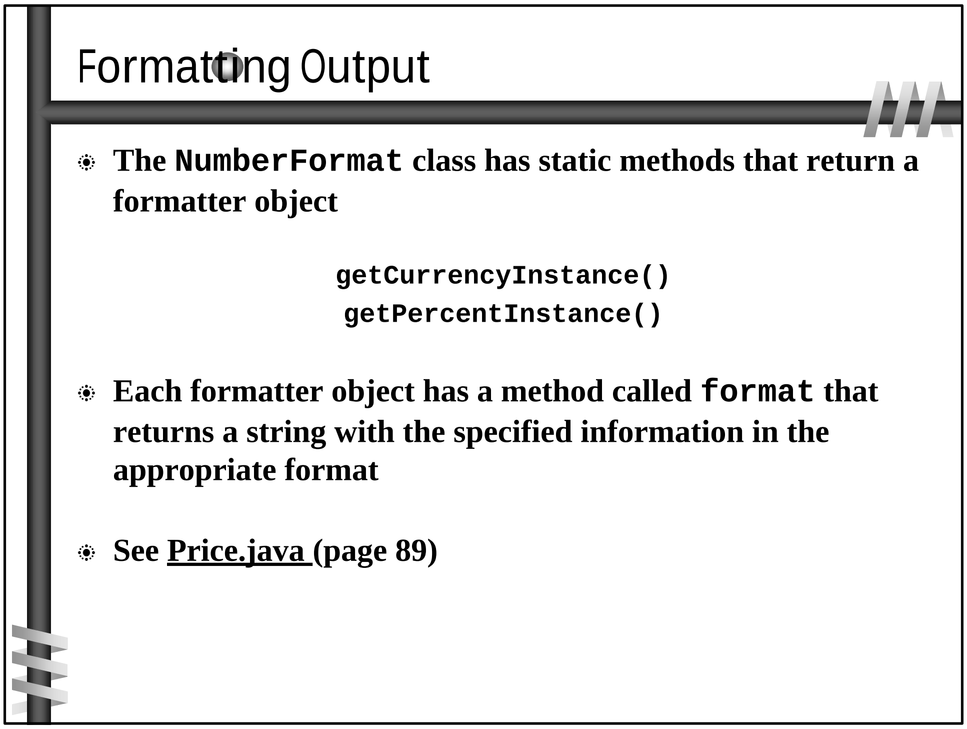
- The `Math` class contains many static methods, providing various mathematical functions, such as absolute value, trigonometry functions, square root, etc.

```
temp = Math.cos(90) + Math.sqrt(delta);
```



The Keyboard Class

- The `Keyboard` class is NOT part of the Java standard class library
- It is provided by the authors of the textbook to make reading input from the keyboard easy
- Details of the `Keyboard` class are explored in Chapter 8
- For now we will simply make use of it
- The `Keyboard` class is part of a package called `cs1`, and contains several static methods for reading particular types of data
- See Echo.java (page 86)
- See Quadratic.java (page 87)

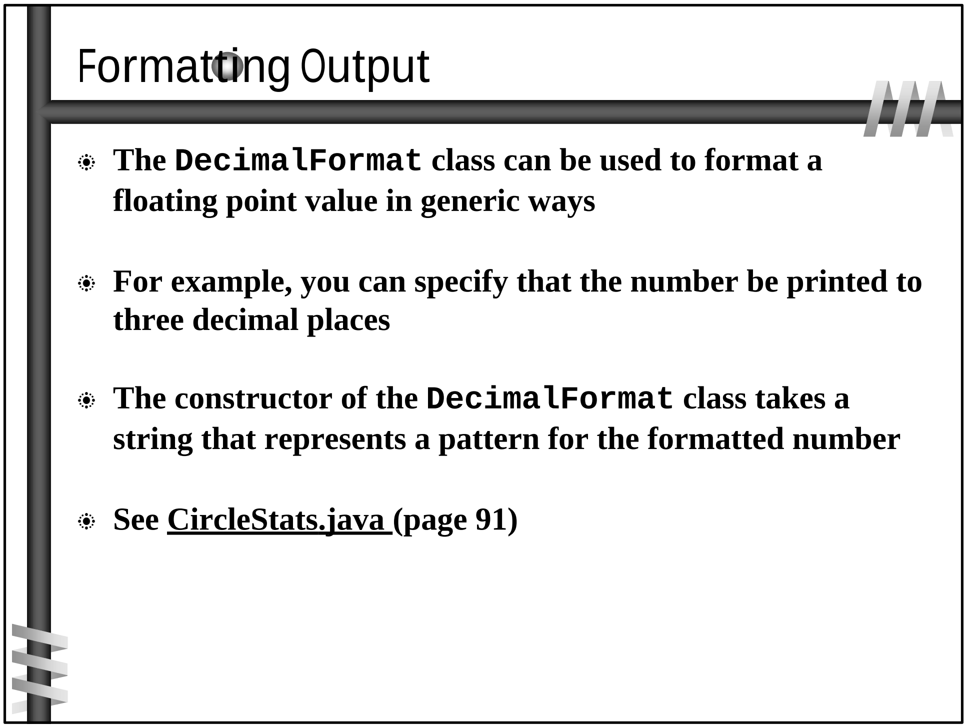


Formatting Output

- The `NumberFormat` class has static methods that return a formatter object

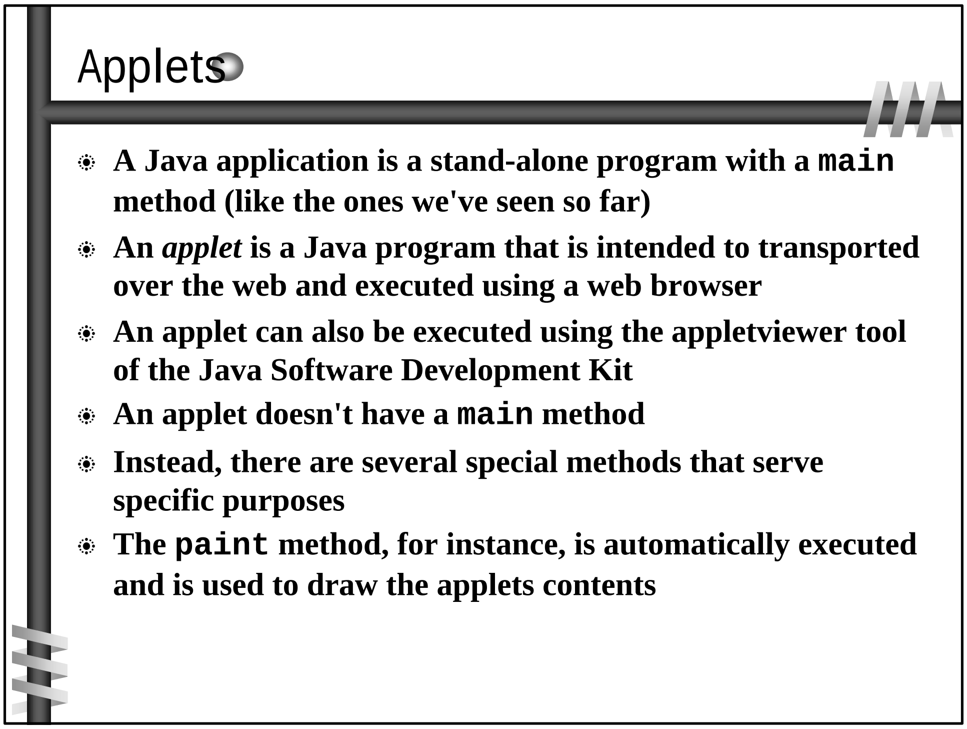
```
getCurrencyInstance()
getPercentInstance()
```

- Each formatter object has a method called `format` that returns a string with the specified information in the appropriate format
- See Price.java (page 89)



Formatting Output

- **The `DecimalFormat` class can be used to format a floating point value in generic ways**
- **For example, you can specify that the number be printed to three decimal places**
- **The constructor of the `DecimalFormat` class takes a string that represents a pattern for the formatted number**
- **See CircleStats.java (page 91)**



Applets

- **A Java application is a stand-alone program with a `main` method (like the ones we've seen so far)**
- **An *applet* is a Java program that is intended to transported over the web and executed using a web browser**
- **An applet can also be executed using the appletviewer tool of the Java Software Development Kit**
- **An applet doesn't have a `main` method**
- **Instead, there are several special methods that serve specific purposes**
- **The `paint` method, for instance, is automatically executed and is used to draw the applets contents**

Applets

- **The paint method accepts a parameter that is an object of the `Graphics` class**
- **A `Graphics` object defines a *graphics context* on which we can draw shapes and text**
- **The `Graphics` class has several methods for drawing shapes**

- **The class that defines the applet *extends* the Applet class**
- **This makes use of *inheritance,* an object-oriented concept explored in more detail in Chapter 7**

- **See Einstein.java (page 93)**

Applets

- **An applet is embedded into an HTML file using a tag that references the bytecode file of the applet class**

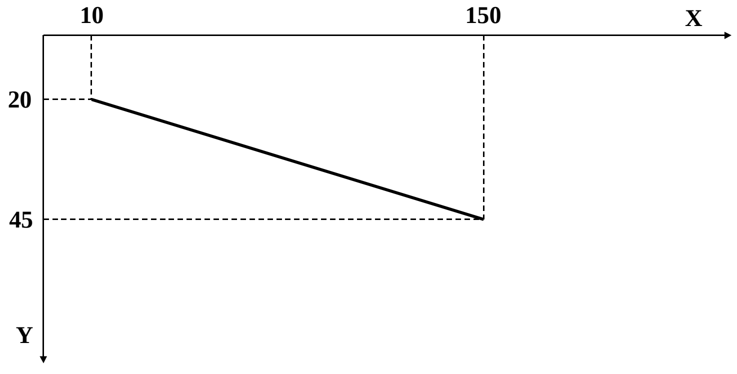
- **It is actually the bytecode version of the program that is transported across the web**

- **The applet is executed by a Java interpreter that is part of the browser**

Drawing Shapes

- Let's explore some of the methods of the `Graphics` class that draw shapes in more detail
- A shape can be filled or unfilled, depending on which method is invoked
- The method parameters specify coordinates and sizes
- Recall from Chapter 1 that the Java coordinate system has the origin in the upper left corner
- Many shapes with curves, like an oval, are drawn by specifying its *bounding rectangle*
- An arc can be thought of as a section of an oval

Drawing a Line

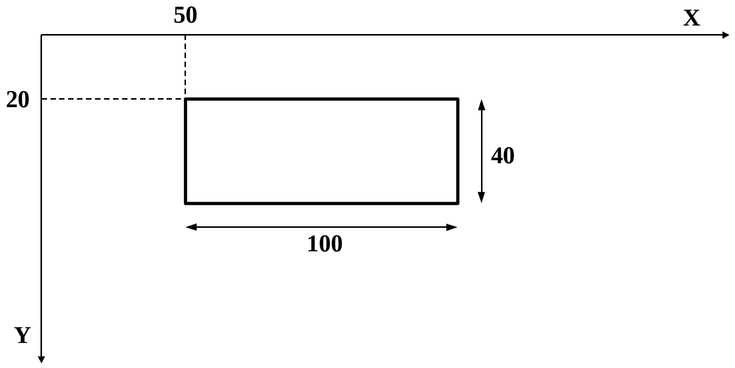


```
page.drawLine (10, 20, 150, 45);
```

or

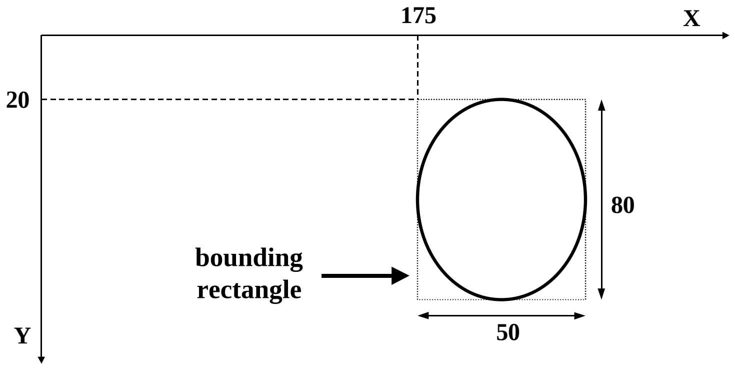
```
page.drawLine (150, 45, 10, 20);
```

Drawing a Rectangle

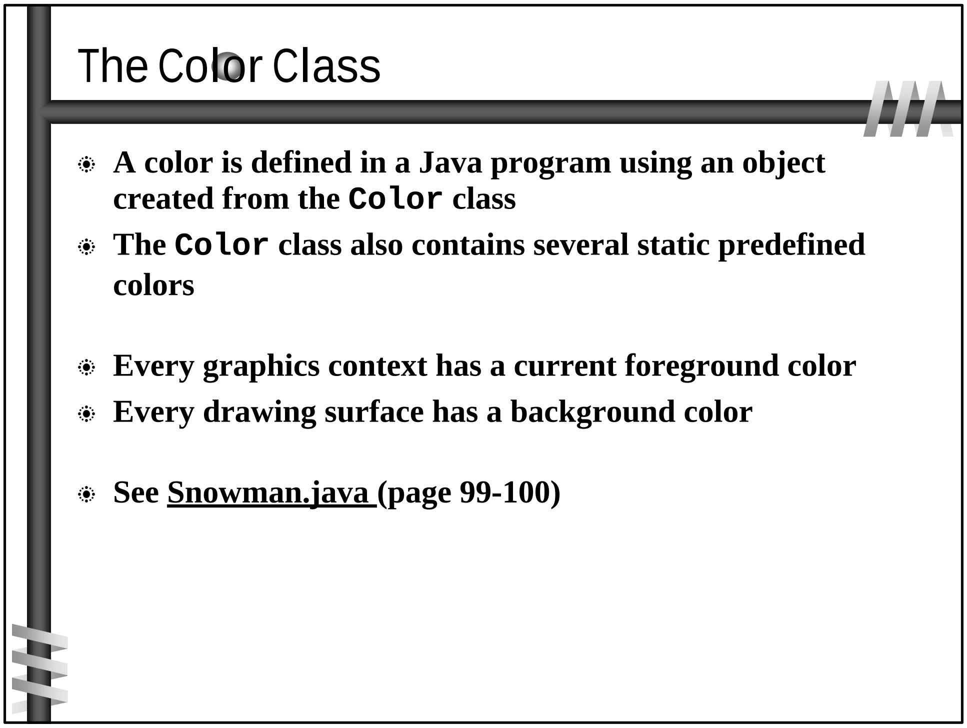


```
page.drawRect (50, 20, 100, 40);
```

Drawing an Oval

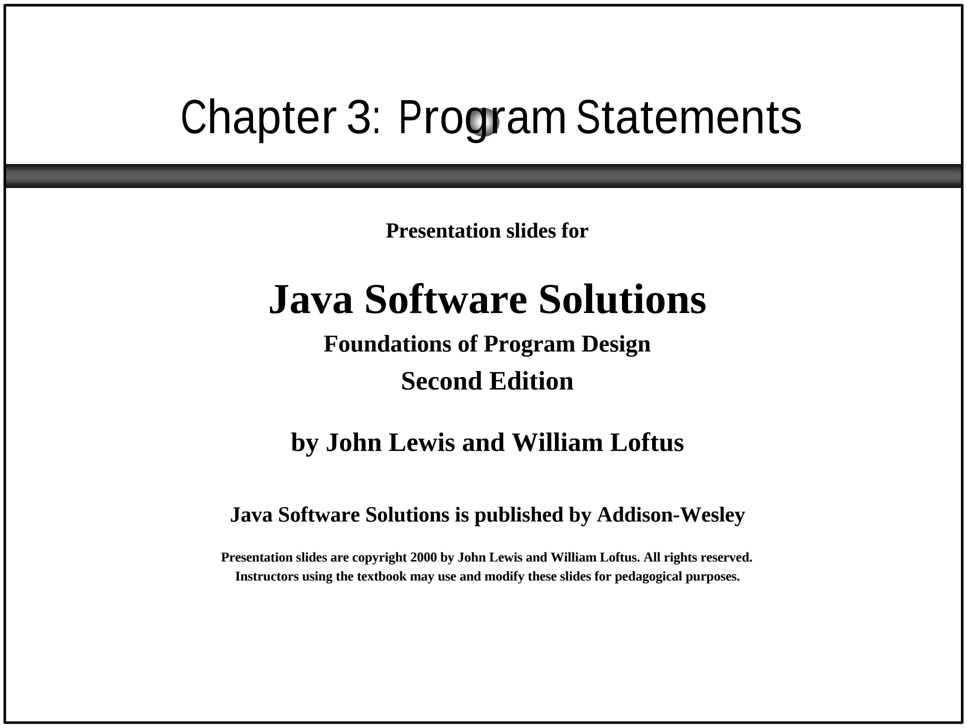


```
page.drawOval (175, 20, 50, 80);
```



The Color Class

- A color is defined in a Java program using an object created from the `Color` class
- The `Color` class also contains several static predefined colors
- Every graphics context has a current foreground color
- Every drawing surface has a background color
- See Snowman.java (page 99-100)



Chapter 3: Program Statements

Presentation slides for

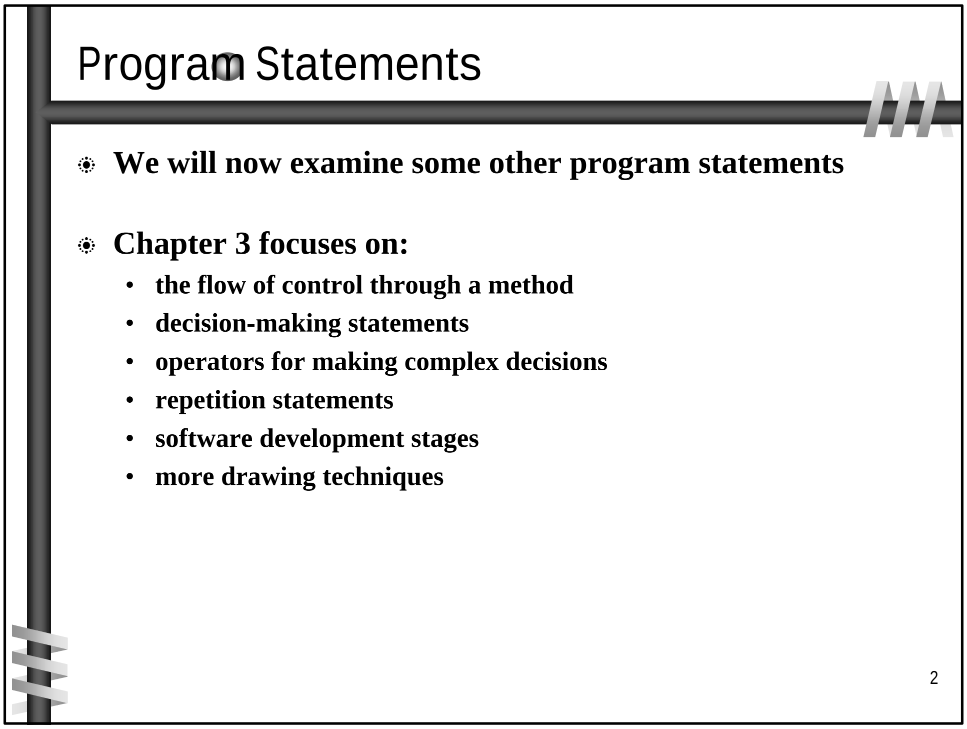
Java Software Solutions

Foundations of Program Design

Second Edition

by John Lewis and William Loftus

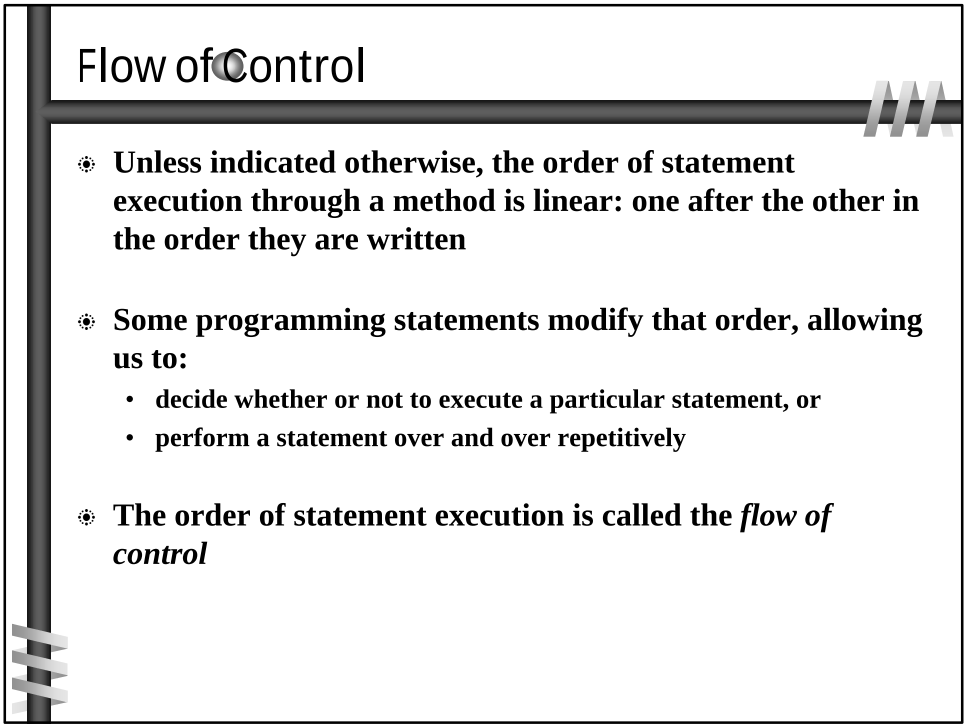
Java Software Solutions is published by Addison-Wesley



Program Statements

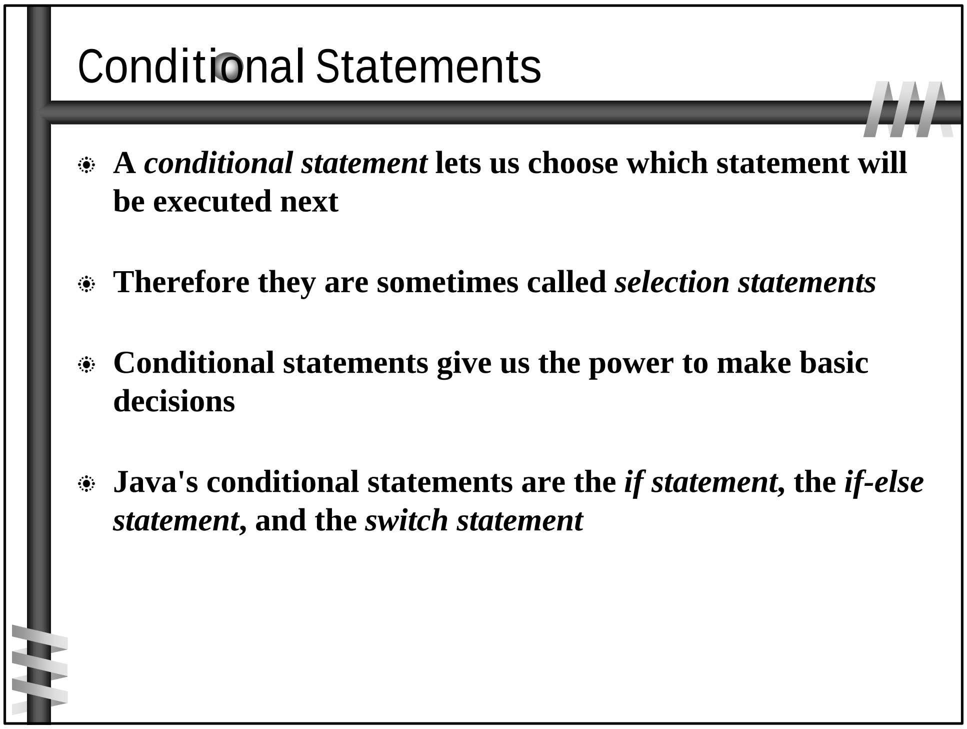
- **We will now examine some other program statements**

- **Chapter 3 focuses on:**
  - **the flow of control through a method**
  - **decision-making statements**
  - **operators for making complex decisions**
  - **repetition statements**
  - **software development stages**
  - **more drawing techniques**



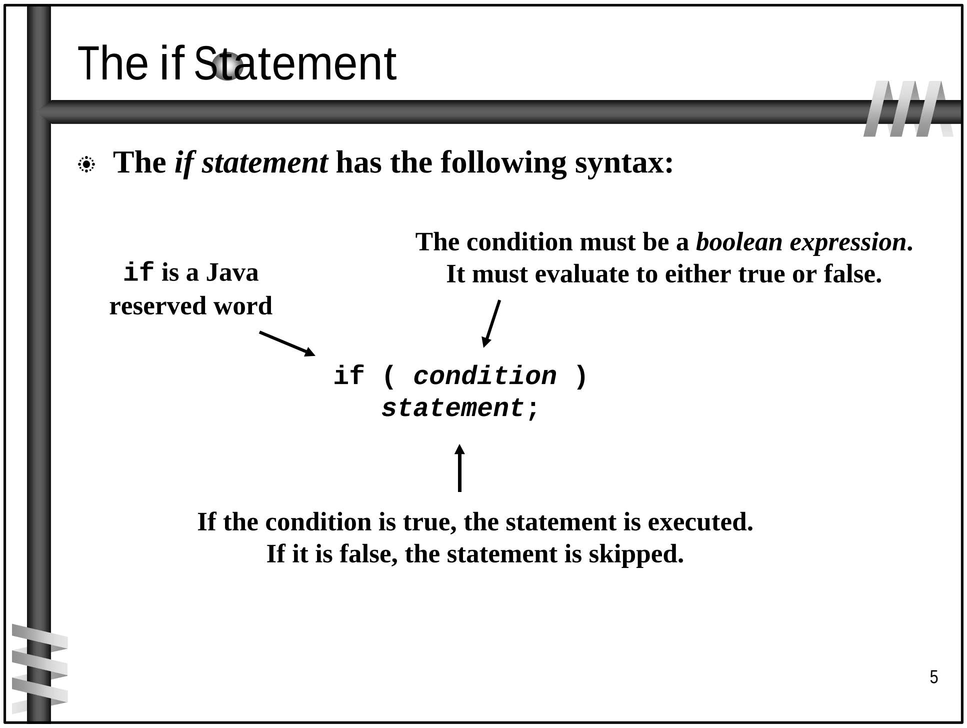
Flow of Control

- Unless indicated otherwise, the order of statement execution through a method is linear: one after the other in the order they are written
- Some programming statements modify that order, allowing us to:
  - decide whether or not to execute a particular statement, or
  - perform a statement over and over repetitively
- The order of statement execution is called the *flow of control*



Conditional Statements

- A *conditional statement* lets us choose which statement will be executed next
- Therefore they are sometimes called *selection statements*
- Conditional statements give us the power to make basic decisions
- Java's conditional statements are the *if statement*, the *if-else statement*, and the *switch statement*



The if Statement

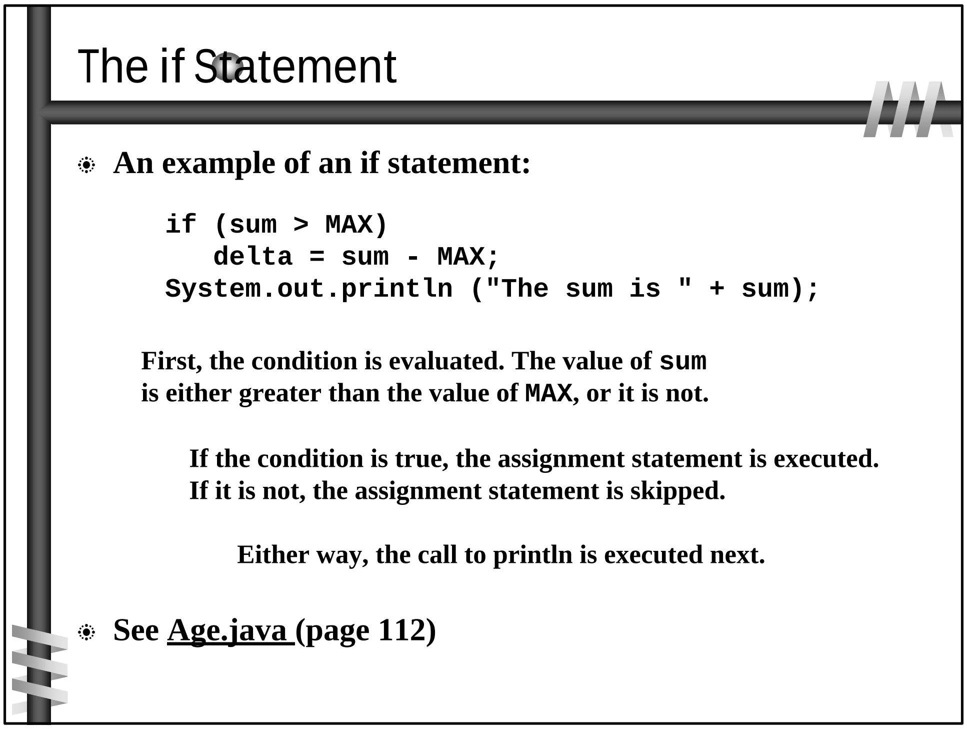
- **The *if statement* has the following syntax:**

`if` **is a Java reserved word**

The condition must be a *boolean expression*. It must evaluate to either true or false.

```
if ( condition )
   statement;
```

If the condition is true, the statement is executed. If it is false, the statement is skipped.



The if Statement

- **An example of an if statement:**

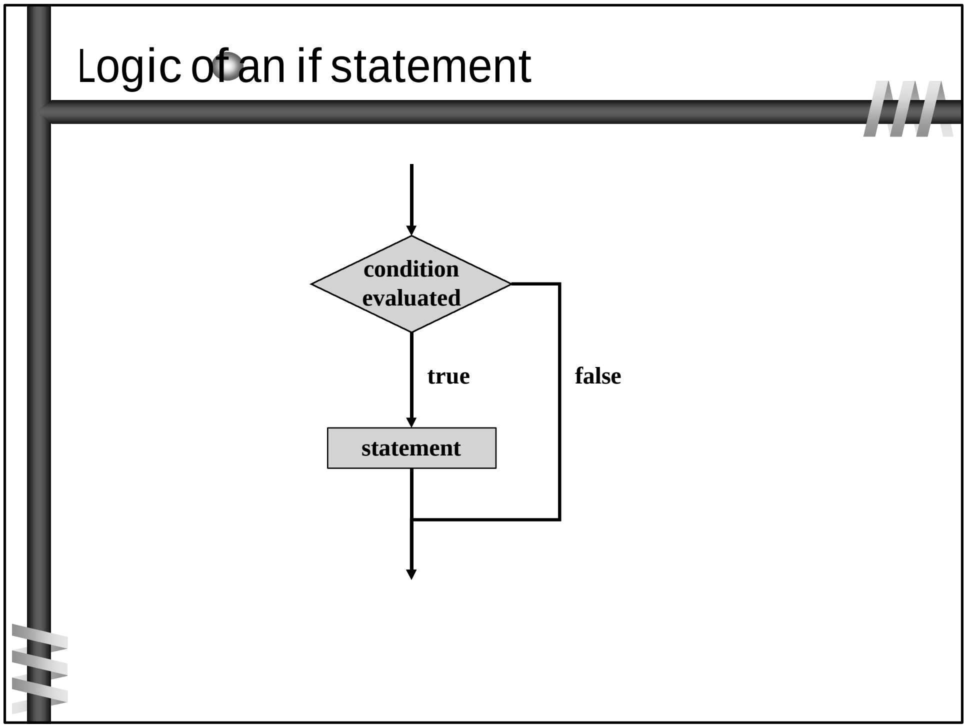
```
if (sum > MAX)
   delta = sum - MAX;
System.out.println ("The sum is " + sum);
```

First, the condition is evaluated. The value of `sum` is either greater than the value of `MAX`, or it is not.

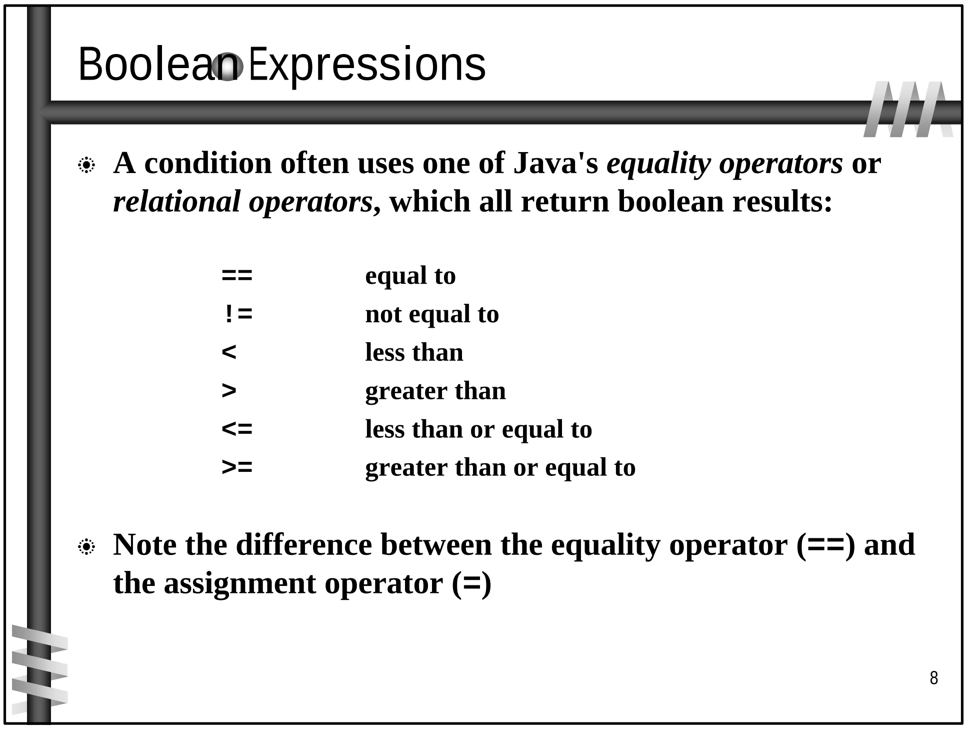
If the condition is true, the assignment statement is executed. If it is not, the assignment statement is skipped.

Either way, the call to println is executed next.

- **See <u>Age.java</u> (page 112)**



Logic of an if statement



Boolean Expressions

- **A condition often uses one of Java's *equality operators* or *relational operators,* which all return boolean results:**

==	**equal to**
!=	**not equal to**
<	**less than**
>	**greater than**
<=	**less than or equal to**
>=	**greater than or equal to**

- **Note the difference between the equality operator (==) and the assignment operator (=)**

8

The if-else Statement

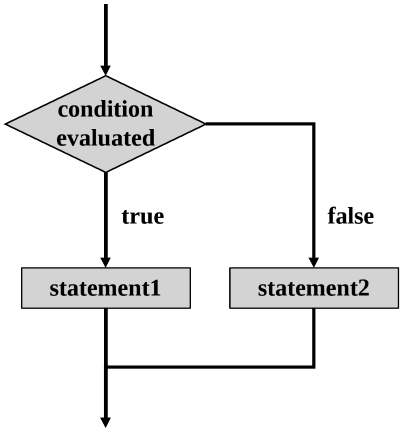
- An *else clause* can be added to an if statement to make it an *if-else statement*:

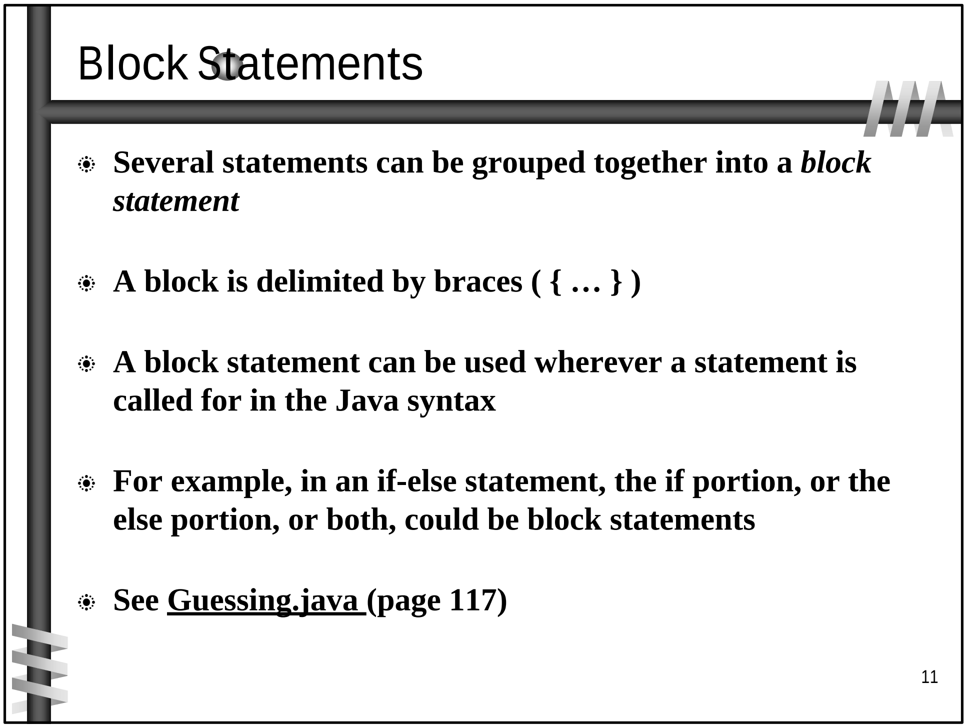
```
if ( condition )
   statement1;
else
   statement2;
```

- If the condition is true, statement1 is executed; if the condition is false, statement2 is executed
- One or the other will be executed, but not both
- See Wages.java (page 116)

9

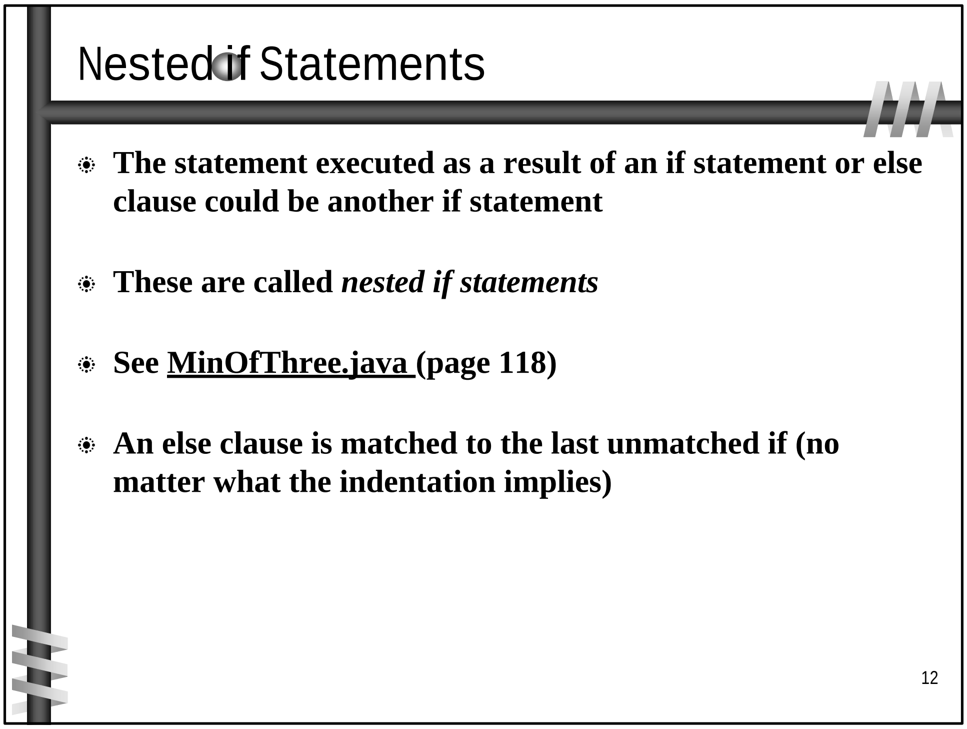
Logic of an if-else statement





Block Statements

- **Several statements can be grouped together into a *block statement***
- **A block is delimited by braces ({ … })**
- **A block statement can be used wherever a statement is called for in the Java syntax**
- **For example, in an if-else statement, the if portion, or the else portion, or both, could be block statements**
- **See <u>Guessing.java</u> (page 117)**



Nested if Statements

- **The statement executed as a result of an if statement or else clause could be another if statement**
- **These are called *nested if statements***
- **See <u>MinOfThree.java</u> (page 118)**
- **An else clause is matched to the last unmatched if (no matter what the indentation implies)**

Comparing Characters

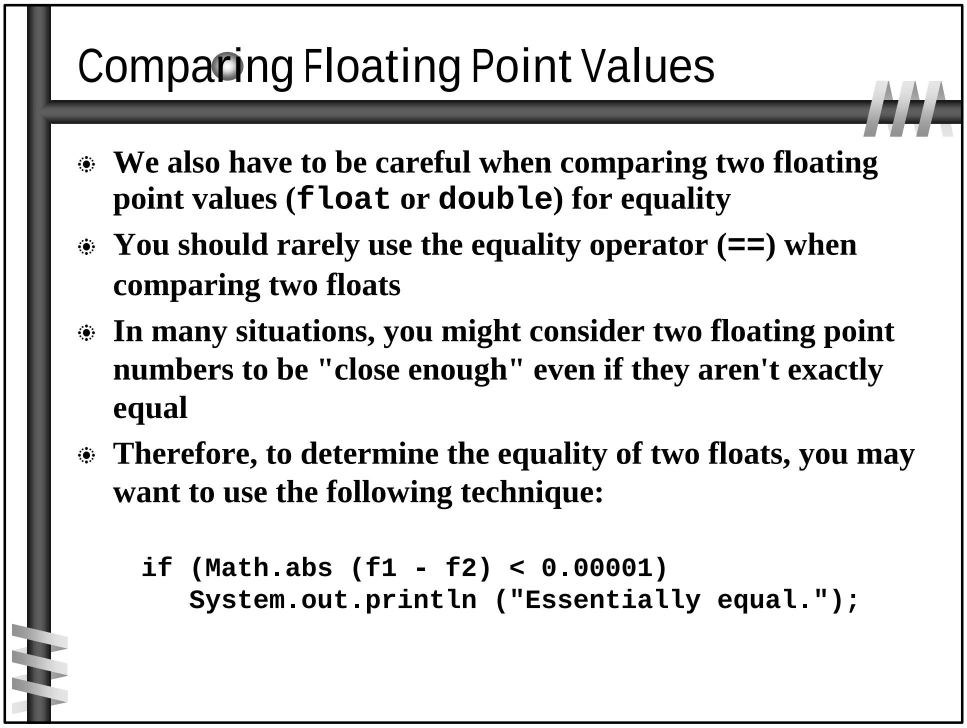
- We can use the relational operators on character data
- The results are based on the Unicode character set
- The following condition is true because the character '+' comes before the character 'J' in Unicode:

```
if ('+' < 'J')
   System.out.println ("+ is less than J");
```

- The uppercase alphabet (A-Z) and the lowercase alphabet (a-z) both appear in alphabetical order in Unicode

Comparing Strings

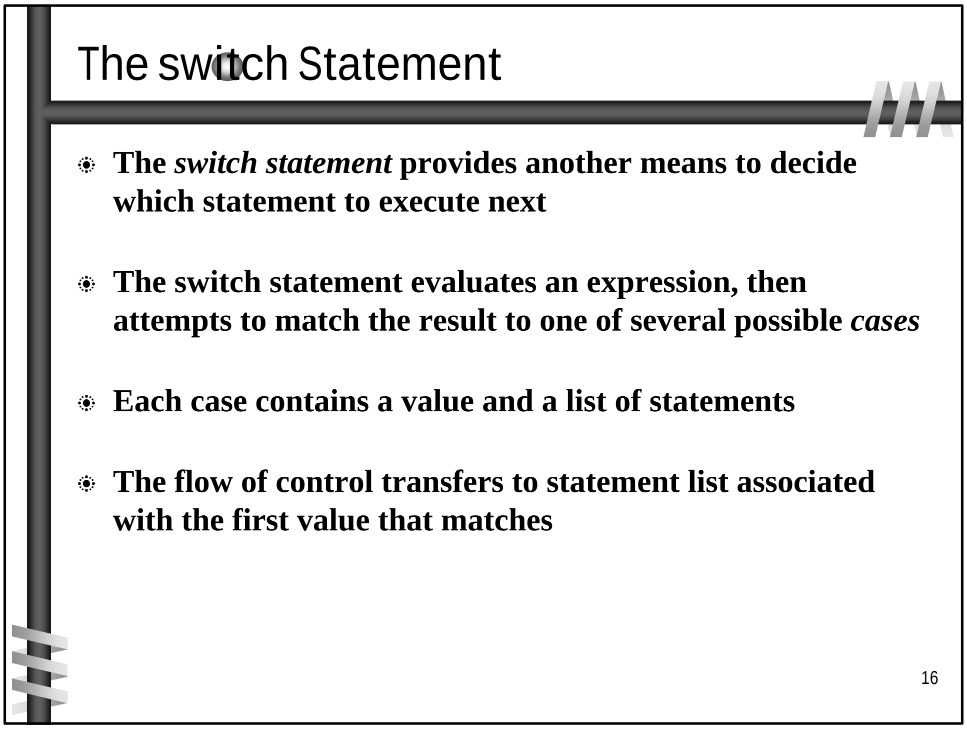
- Remember that a character string in Java is an object
- We cannot use the relational operators to compare strings
- The `equals` method can be called on a string to determine if two strings contain exactly the same characters in the same order
- The String class also contains a method called `compareTo` to determine if one string comes before another alphabetically (as determined by the Unicode character set)



Comparing Floating Point Values

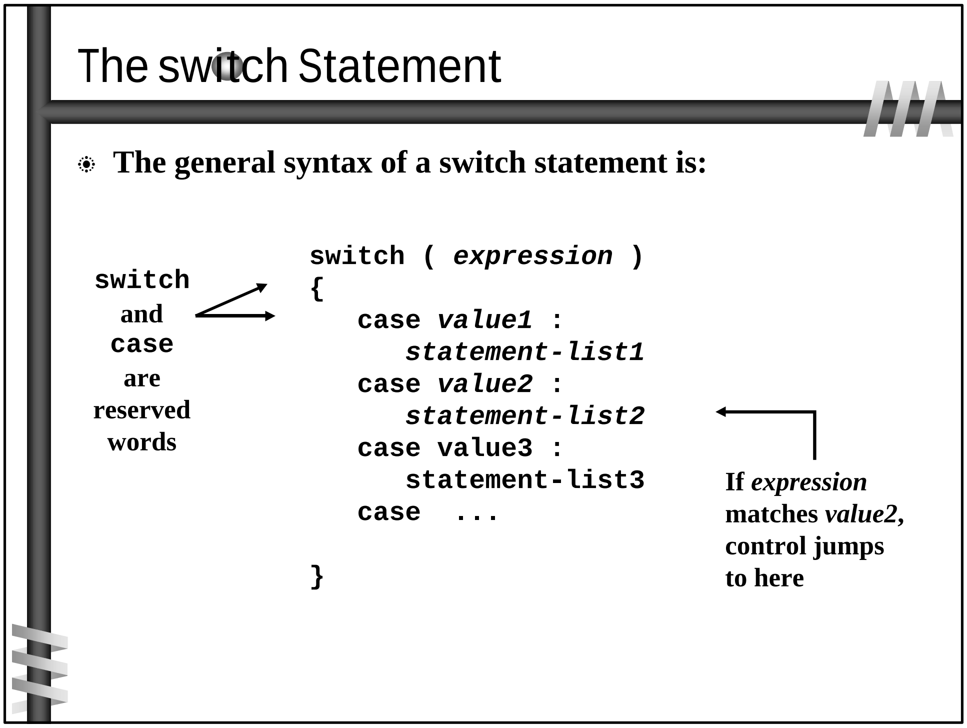
- We also have to be careful when comparing two floating point values (`float` or `double`) for equality
- You should rarely use the equality operator (`==`) when comparing two floats
- In many situations, you might consider two floating point numbers to be "close enough" even if they aren't exactly equal
- Therefore, to determine the equality of two floats, you may want to use the following technique:

```
if (Math.abs (f1 - f2) < 0.00001)
   System.out.println ("Essentially equal.");
```



The switch Statement

- The *switch statement* provides another means to decide which statement to execute next
- The switch statement evaluates an expression, then attempts to match the result to one of several possible *cases*
- Each case contains a value and a list of statements
- The flow of control transfers to statement list associated with the first value that matches



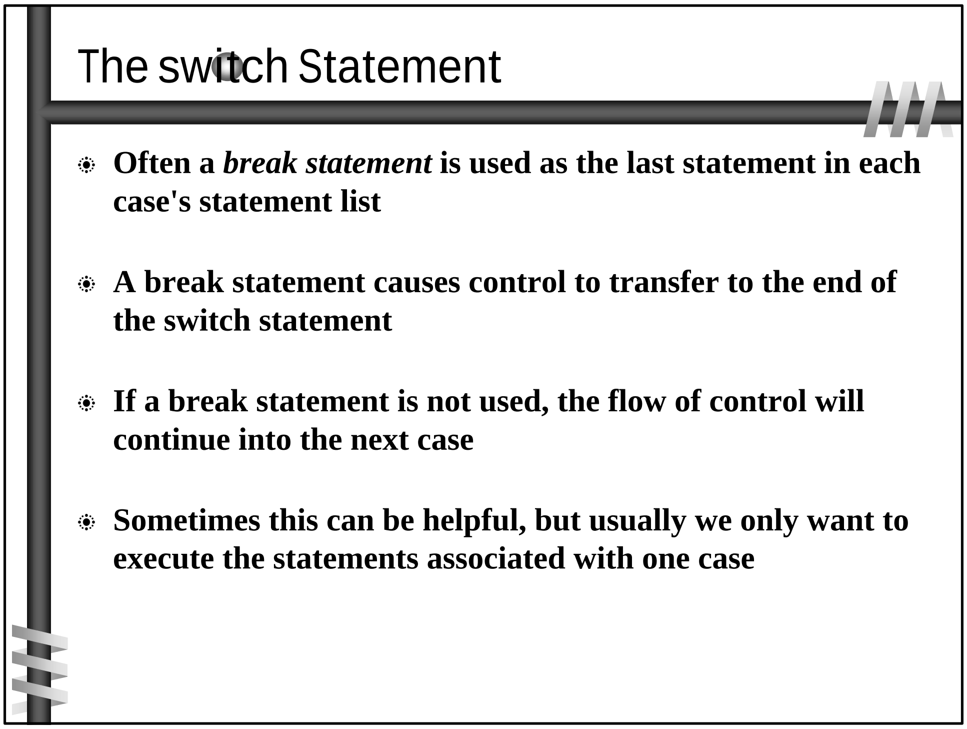
The switch Statement

- The general syntax of a switch statement is:

switch
and
case
are
reserved
words

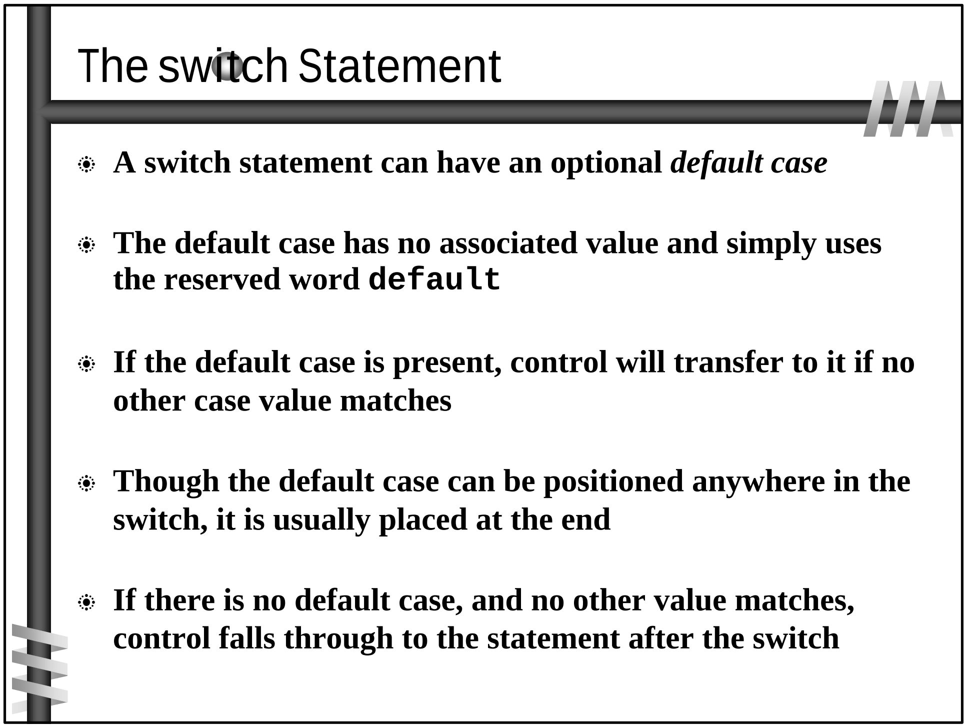
```
switch ( expression )
{
   case value1 :
      statement-list1
   case value2 :
      statement-list2
   case value3 :
      statement-list3
   case  ...

}
```

If *expression* matches *value2*, control jumps to here



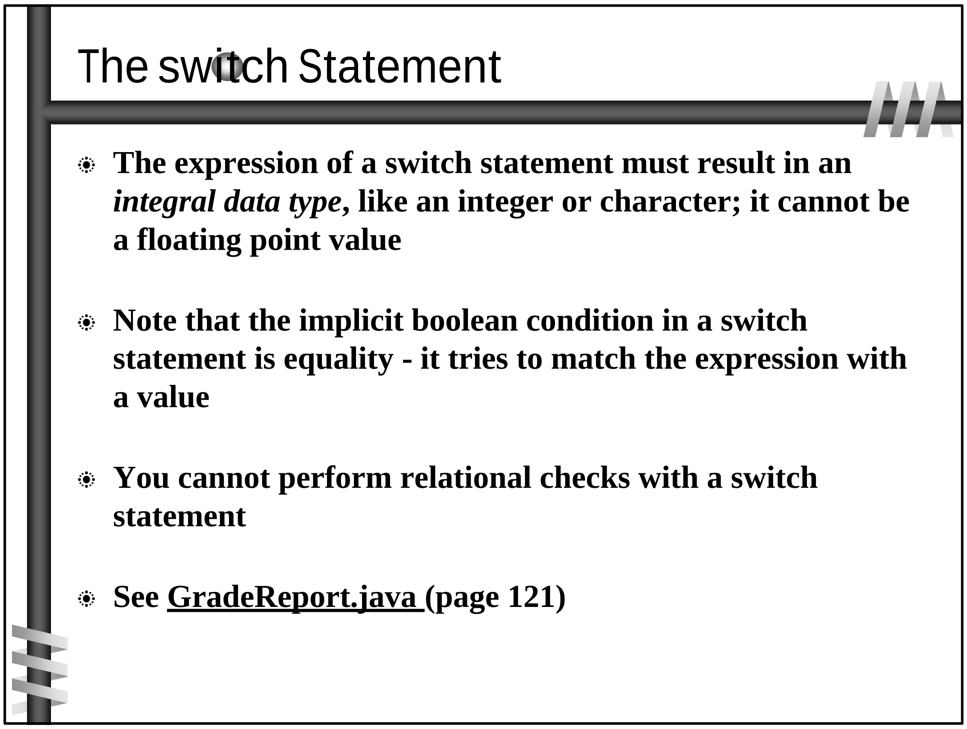
The switch Statement

- Often a *break statement* is used as the last statement in each case's statement list
- A break statement causes control to transfer to the end of the switch statement
- If a break statement is not used, the flow of control will continue into the next case
- Sometimes this can be helpful, but usually we only want to execute the statements associated with one case



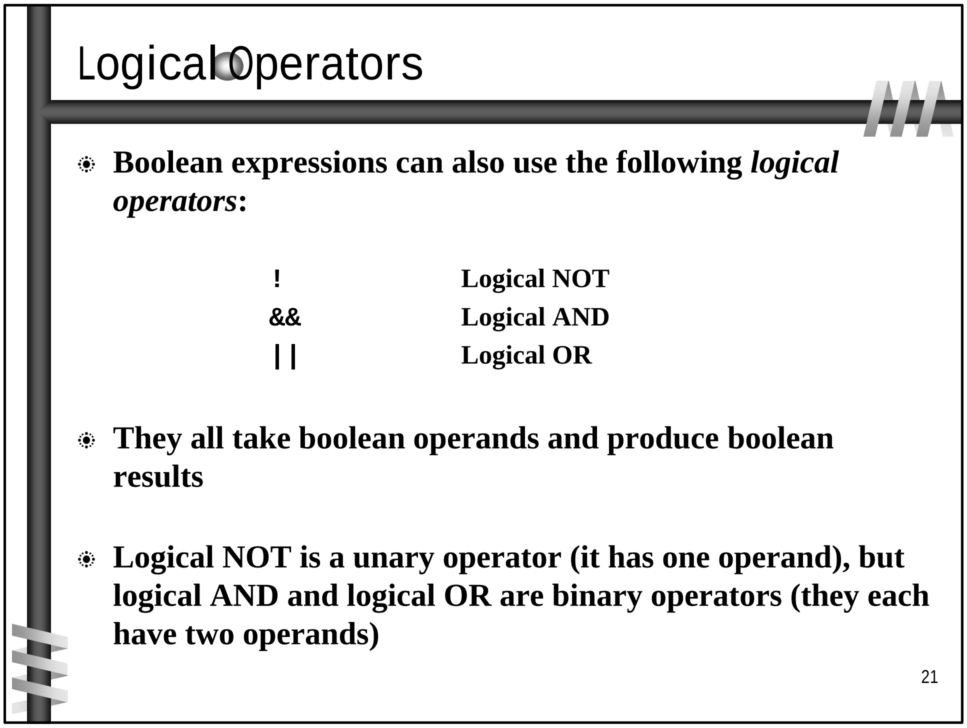
The switch Statement

- A switch statement can have an optional *default case*
- The default case has no associated value and simply uses the reserved word `default`
- If the default case is present, control will transfer to it if no other case value matches
- Though the default case can be positioned anywhere in the switch, it is usually placed at the end
- If there is no default case, and no other value matches, control falls through to the statement after the switch



The switch Statement

- The expression of a switch statement must result in an *integral data type*, like an integer or character; it cannot be a floating point value
- Note that the implicit boolean condition in a switch statement is equality - it tries to match the expression with a value
- You cannot perform relational checks with a switch statement
- See <u>GradeReport.java</u> (page 121)

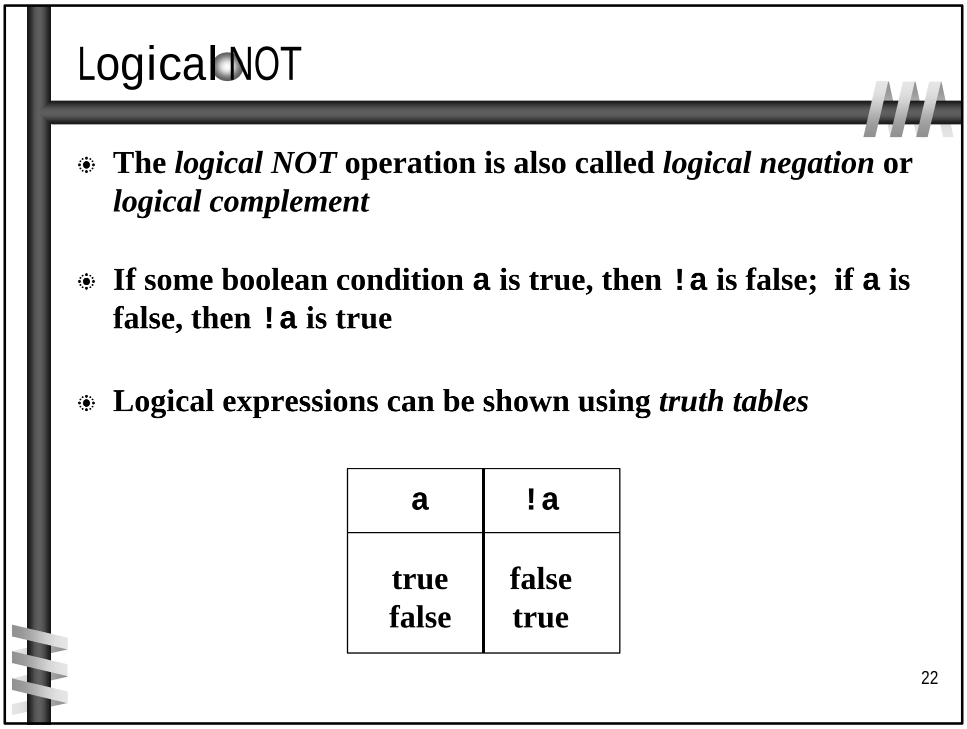


Logical Operators

- Boolean expressions can also use the following *logical operators*:

!	Logical NOT
&&	Logical AND
||	Logical OR

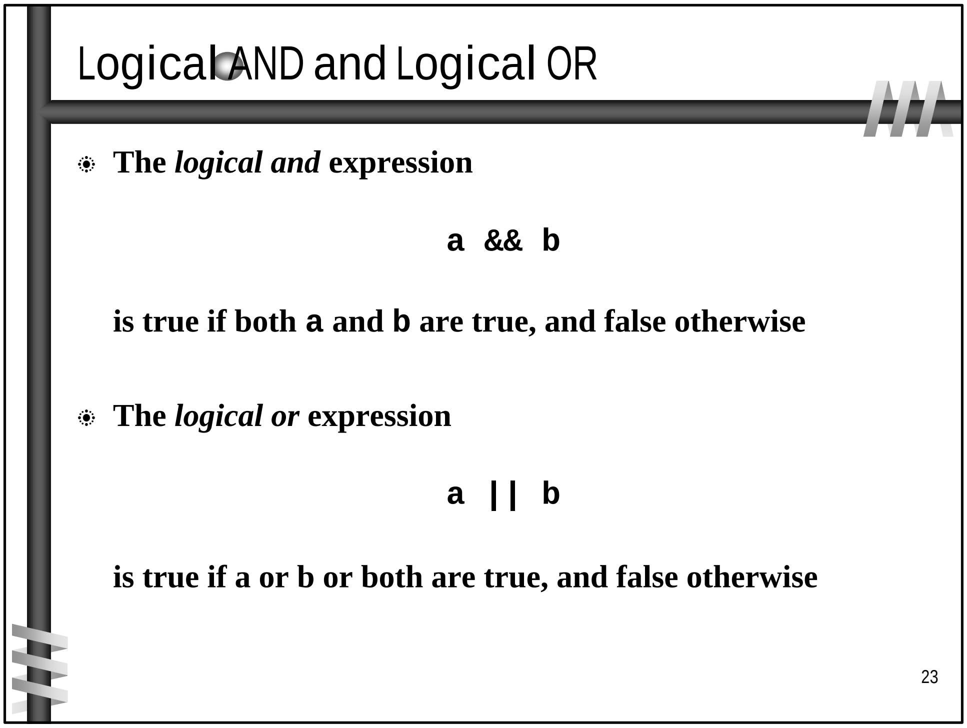
- They all take boolean operands and produce boolean results
- Logical NOT is a unary operator (it has one operand), but logical AND and logical OR are binary operators (they each have two operands)



Logical NOT

- The *logical NOT* operation is also called *logical negation* or *logical complement*
- If some boolean condition `a` is true, then `!a` is false; if `a` is false, then `!a` is true
- Logical expressions can be shown using *truth tables*

a	!a
true false	false true



Logical AND and Logical OR

- The *logical and* expression

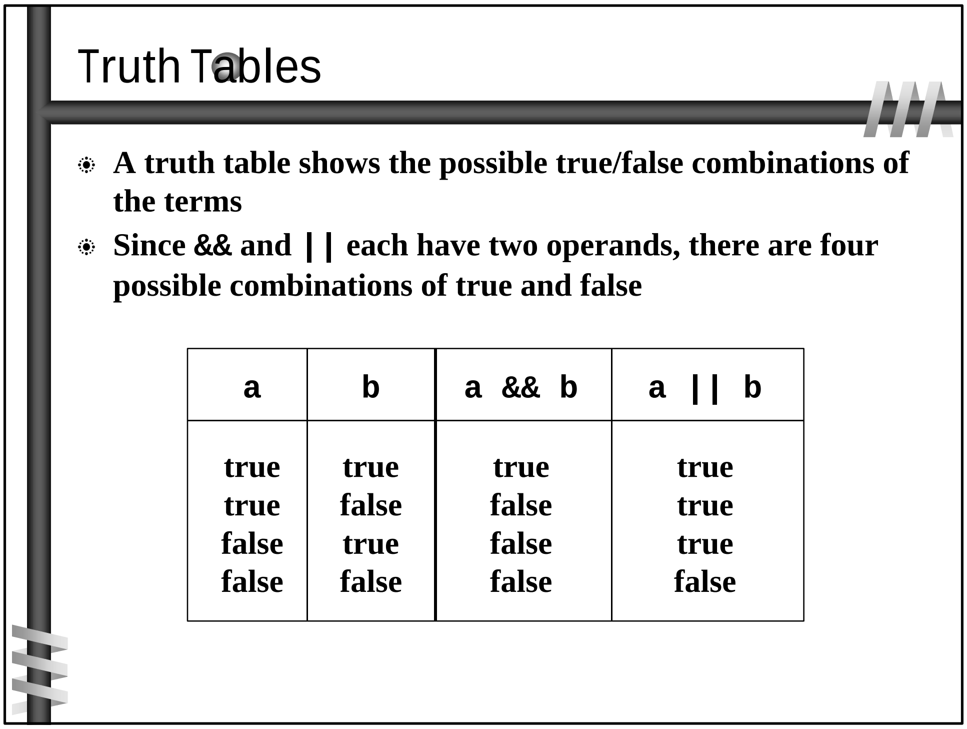
  a && b

  is true if both a and b are true, and false otherwise

- The *logical or* expression

  a || b

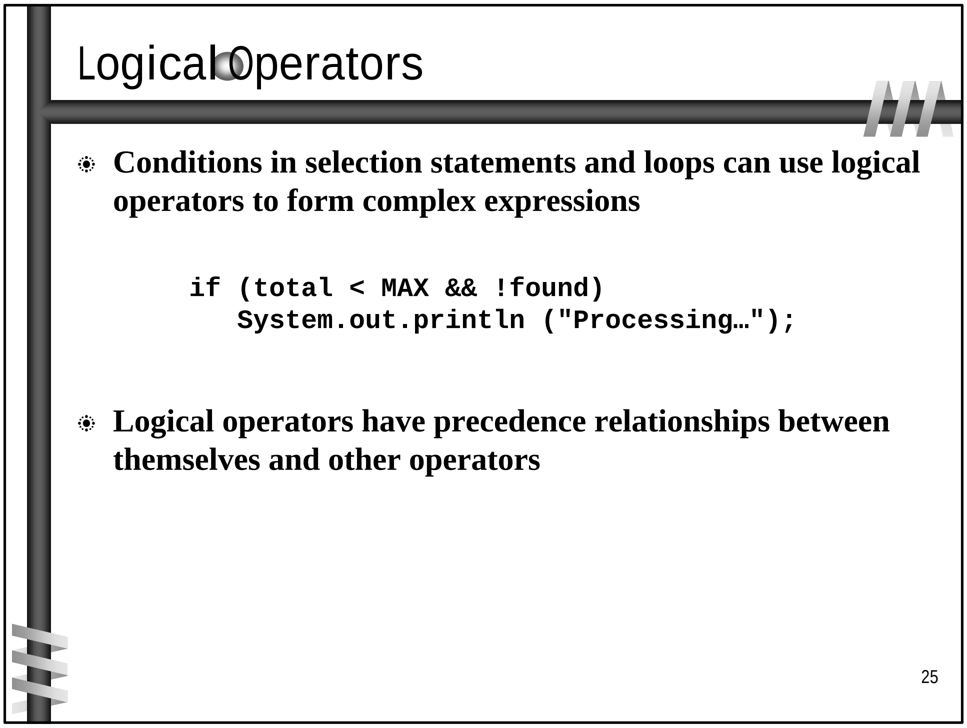
  is true if a or b or both are true, and false otherwise



Truth Tables

- A truth table shows the possible true/false combinations of the terms
- Since && and || each have two operands, there are four possible combinations of true and false

a	b	a && b	a || b
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

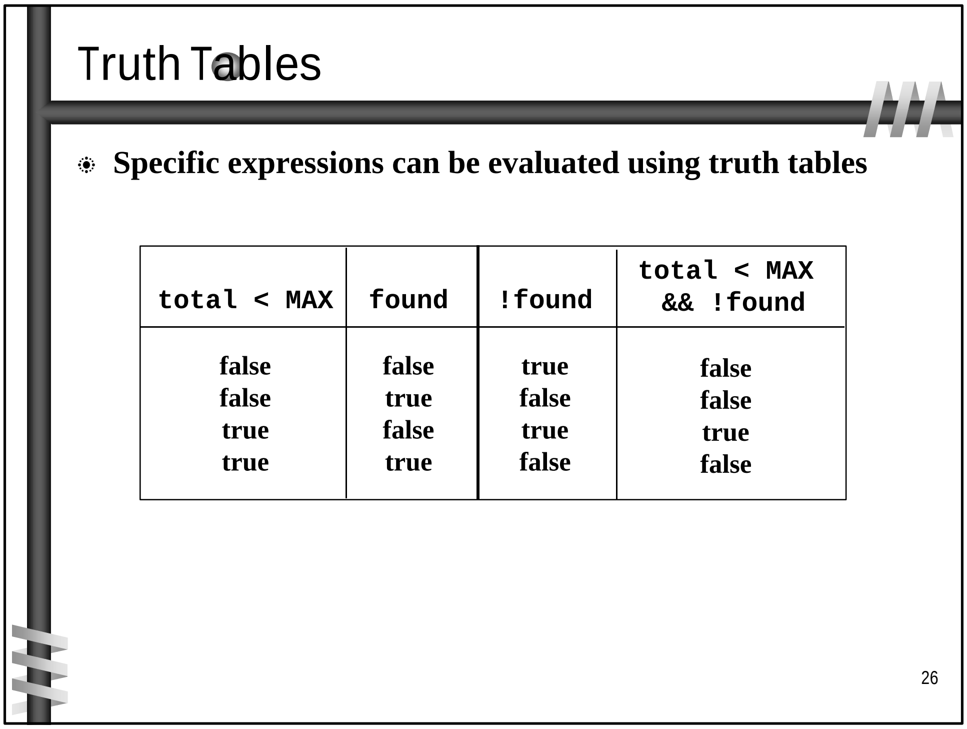


Logical Operators

- Conditions in selection statements and loops can use logical operators to form complex expressions

```
if (total < MAX && !found)
   System.out.println ("Processing…");
```

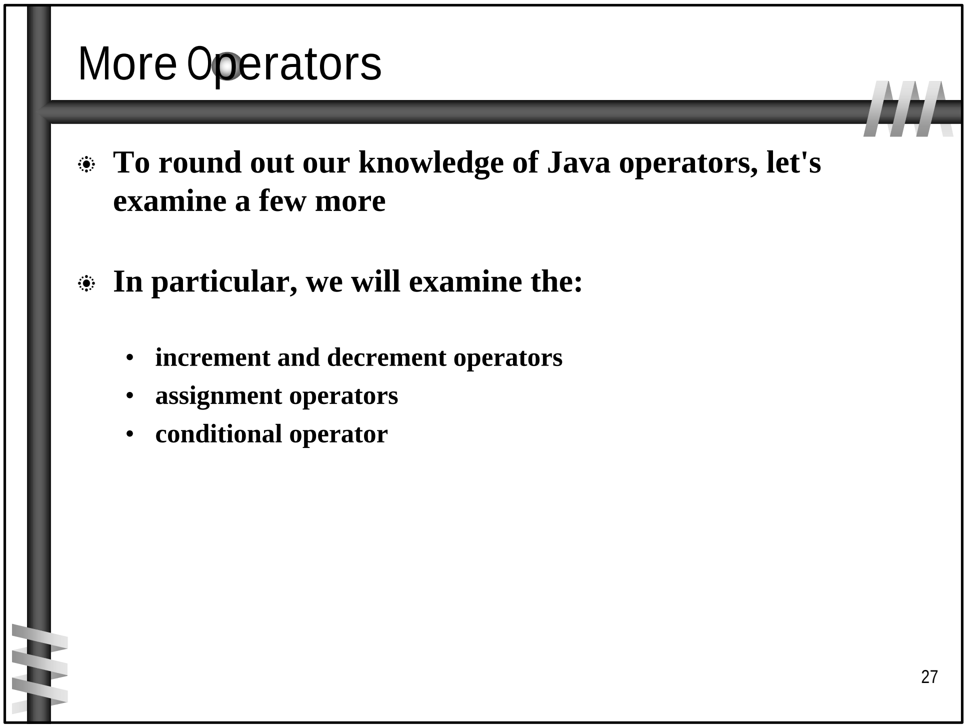
- Logical operators have precedence relationships between themselves and other operators



Truth Tables

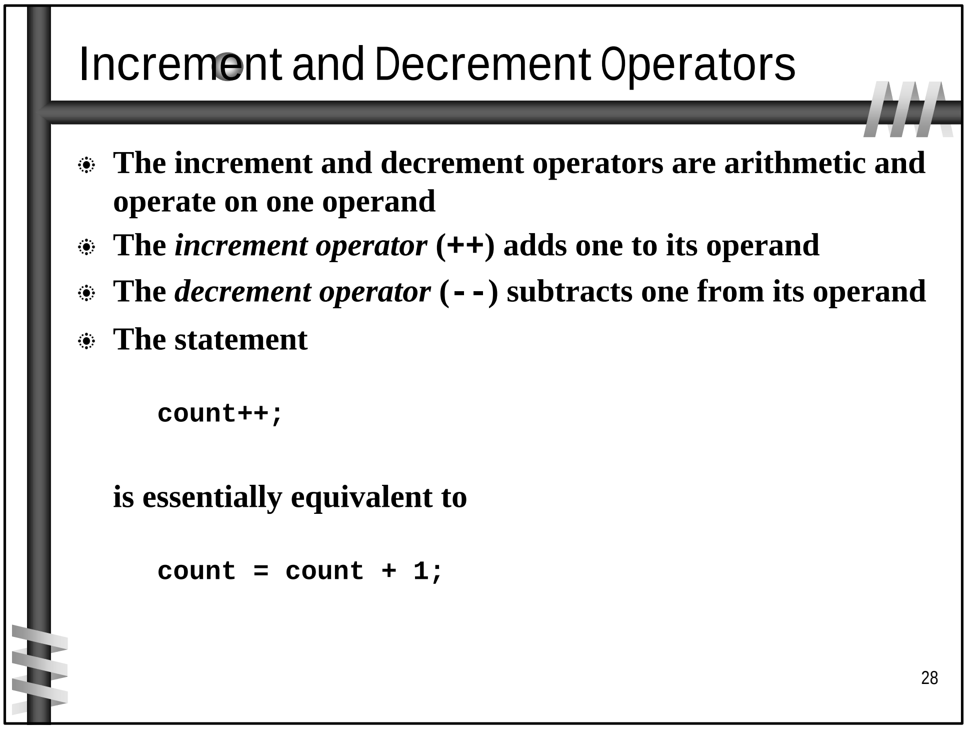
- Specific expressions can be evaluated using truth tables

total < MAX	found	!found	total < MAX && !found
false	false	true	false
false	true	false	false
true	false	true	true
true	true	false	false



More Operators

- To round out our knowledge of Java operators, let's examine a few more
- In particular, we will examine the:
  - increment and decrement operators
  - assignment operators
  - conditional operator



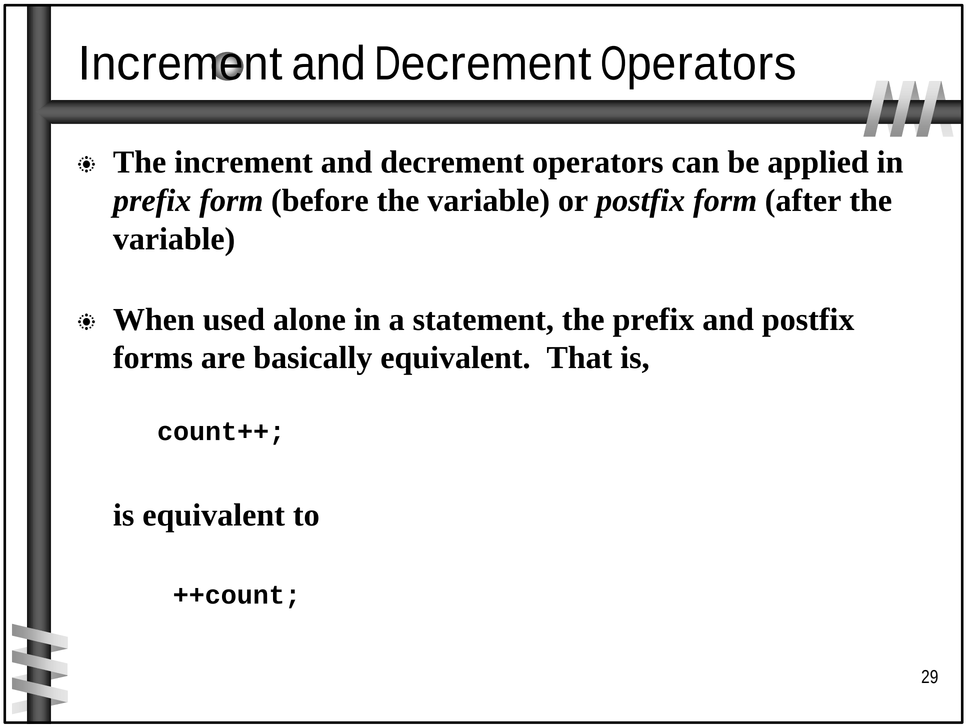
Increment and Decrement Operators

- The increment and decrement operators are arithmetic and operate on one operand
- The *increment operator* (`++`) adds one to its operand
- The *decrement operator* (`--`) subtracts one from its operand
- The statement

```
count++;
```

is essentially equivalent to

```
count = count + 1;
```



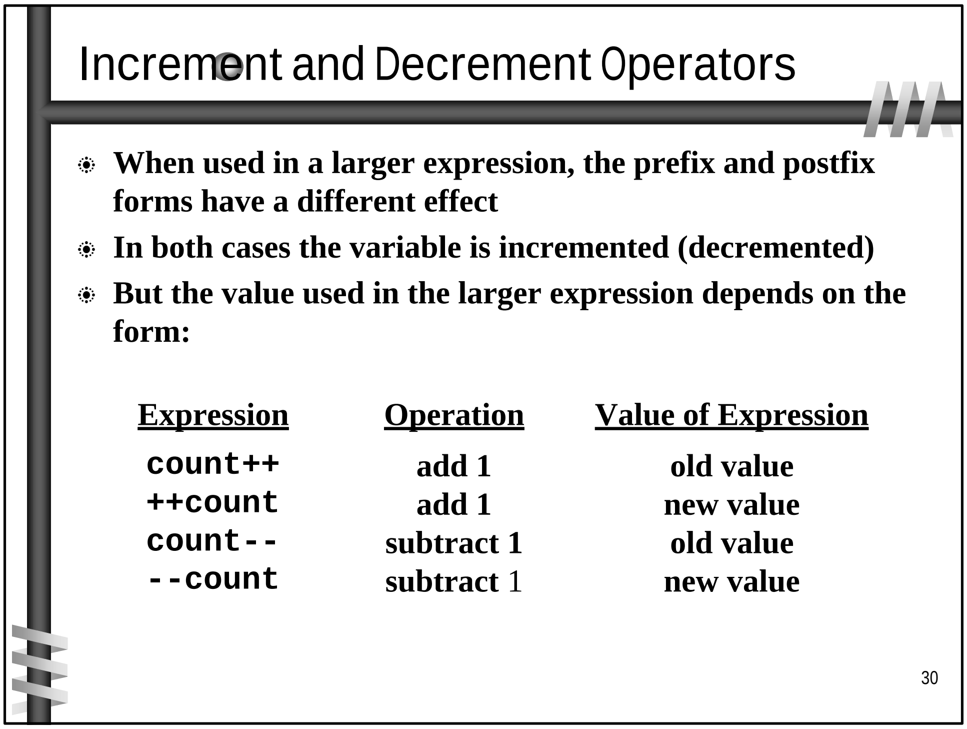
Increment and Decrement Operators

- **The increment and decrement operators can be applied in *prefix form* (before the variable) or *postfix form* (after the variable)**
- **When used alone in a statement, the prefix and postfix forms are basically equivalent. That is,**

```
count++;
```

is equivalent to

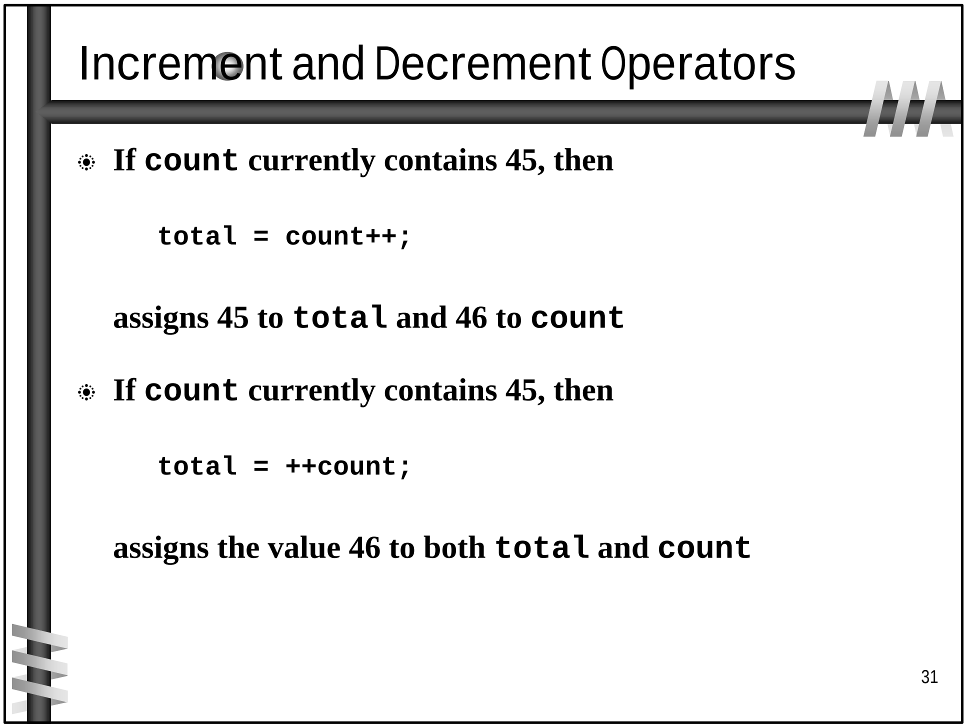
```
++count;
```



Increment and Decrement Operators

- **When used in a larger expression, the prefix and postfix forms have a different effect**
- **In both cases the variable is incremented (decremented)**
- **But the value used in the larger expression depends on the form:**

Expression	**Operation**	**Value of Expression**
`count++`	**add 1**	**old value**
`++count`	**add 1**	**new value**
`count--`	**subtract 1**	**old value**
`--count`	**subtract** 1	**new value**



Increment and Decrement Operators

- If **`count`** currently contains 45, then

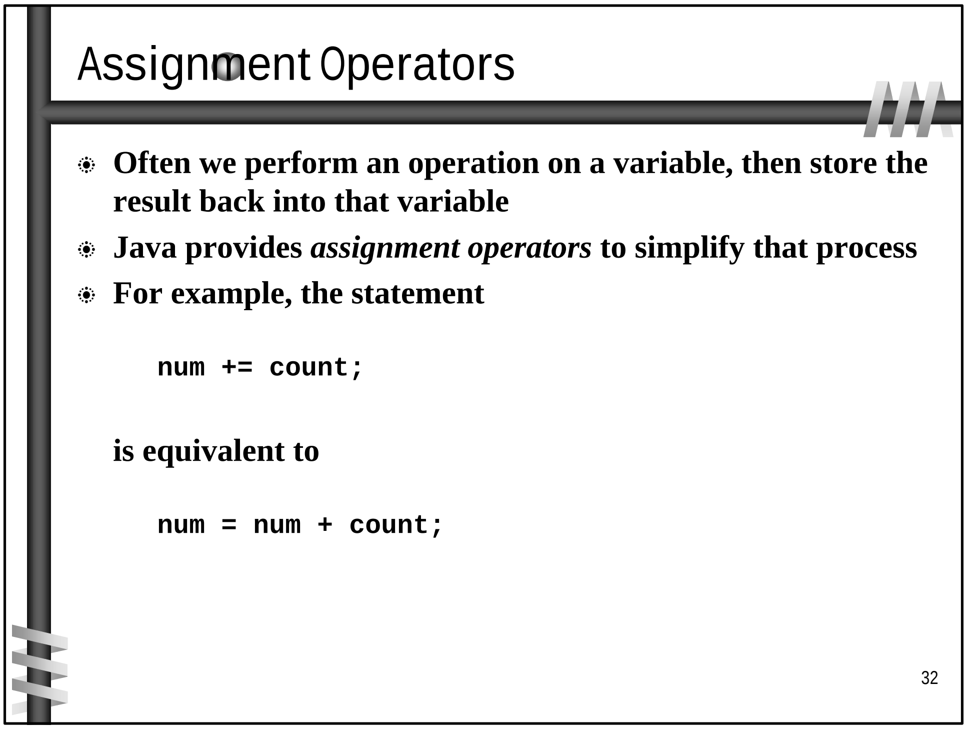
```
total = count++;
```

assigns 45 to `total` and 46 to `count`

- If **`count`** currently contains 45, then

```
total = ++count;
```

assigns the value 46 to both `total` and `count`



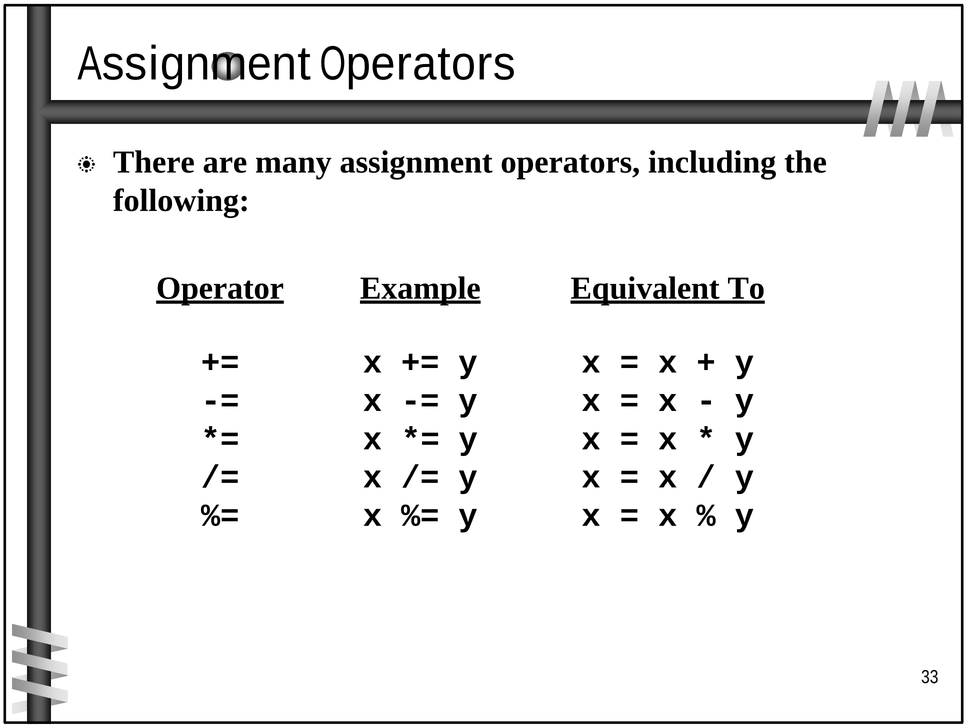
Assignment Operators

- **Often we perform an operation on a variable, then store the result back into that variable**
- **Java provides *assignment operators* to simplify that process**
- **For example, the statement**

```
num += count;
```

is equivalent to

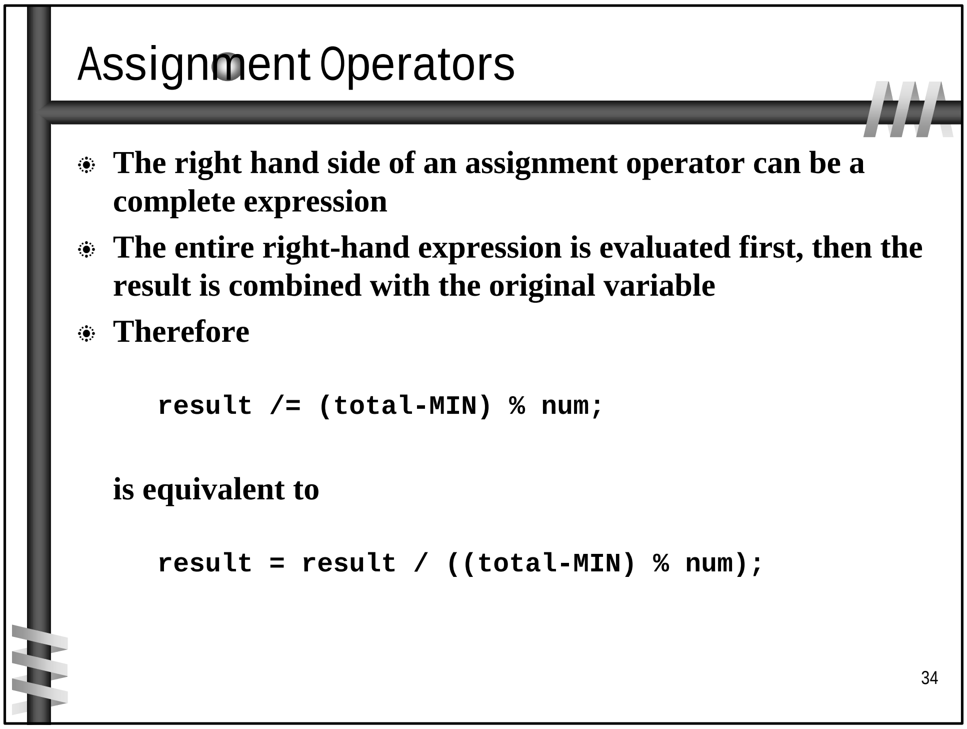
```
num = num + count;
```



Assignment Operators

- There are many assignment operators, including the following:

Operator	Example	Equivalent To
`+=`	`x += y`	`x = x + y`
`-=`	`x -= y`	`x = x - y`
`*=`	`x *= y`	`x = x * y`
`/=`	`x /= y`	`x = x / y`
`%=`	`x %= y`	`x = x % y`



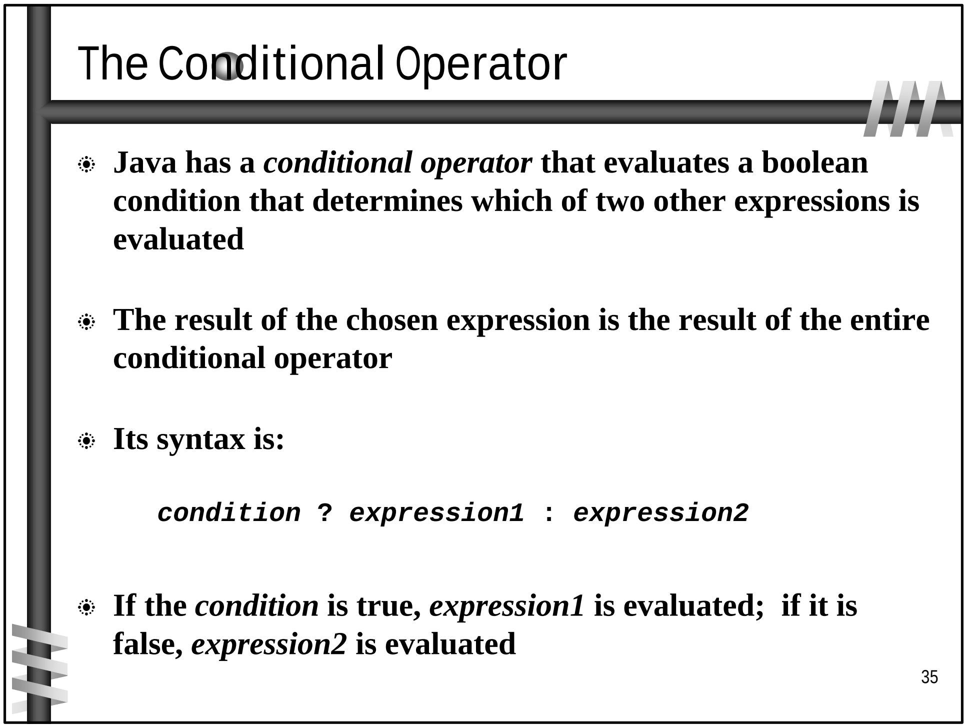
Assignment Operators

- The right hand side of an assignment operator can be a complete expression
- The entire right-hand expression is evaluated first, then the result is combined with the original variable
- Therefore

```
result /= (total-MIN) % num;
```

is equivalent to

```
result = result / ((total-MIN) % num);
```

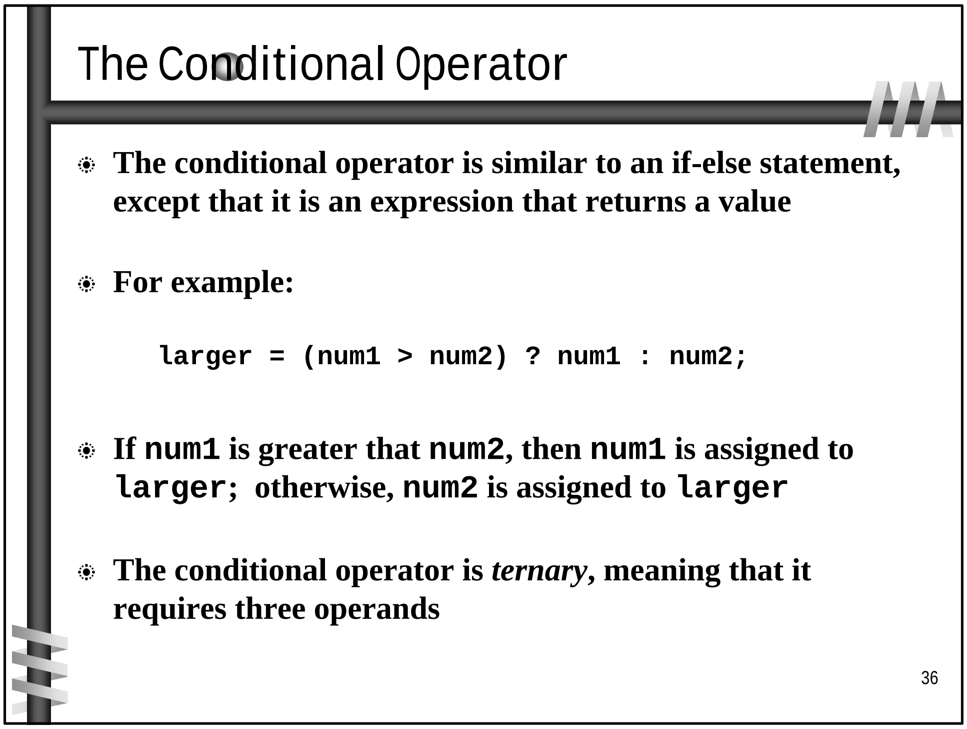


The Conditional Operator

- Java has a *conditional operator* that evaluates a boolean condition that determines which of two other expressions is evaluated
- The result of the chosen expression is the result of the entire conditional operator
- Its syntax is:

```
condition ? expression1 : expression2
```

- If the *condition* is true, *expression1* is evaluated; if it is false, *expression2* is evaluated

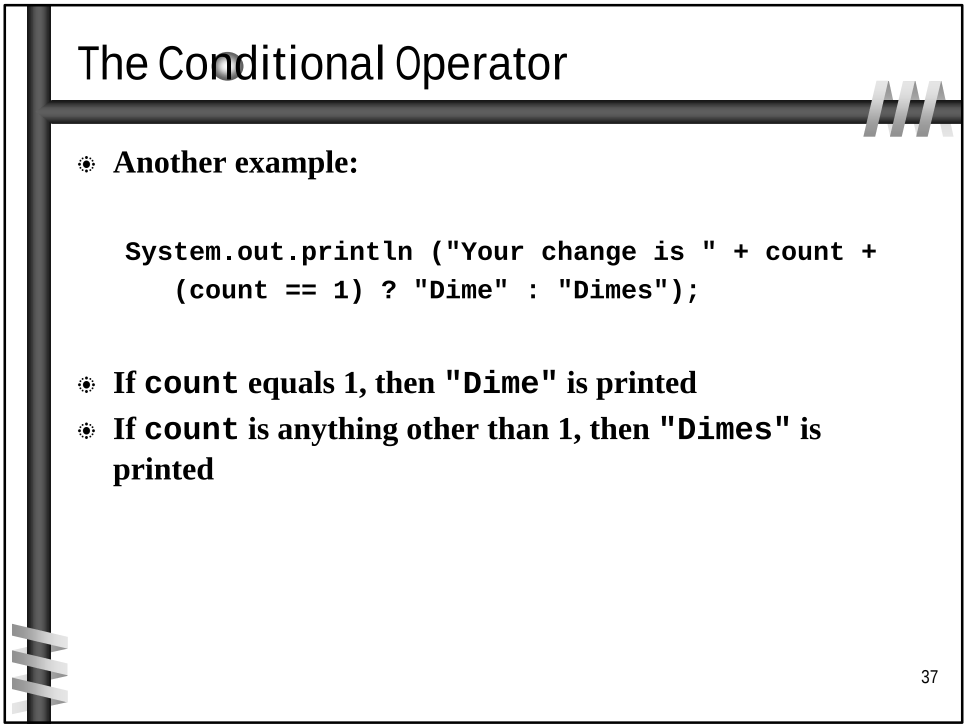


The Conditional Operator

- The conditional operator is similar to an if-else statement, except that it is an expression that returns a value
- For example:

```
larger = (num1 > num2) ? num1 : num2;
```

- If `num1` is greater that `num2`, then `num1` is assigned to `larger`; otherwise, `num2` is assigned to `larger`
- The conditional operator is *ternary*, meaning that it requires three operands



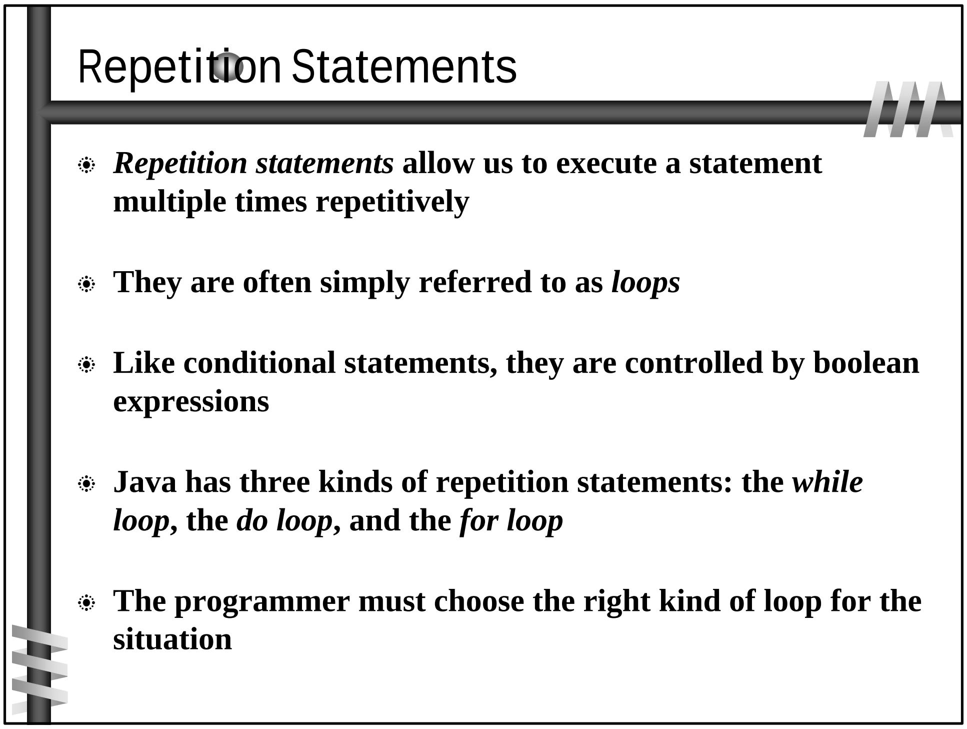
The Conditional Operator

- **Another example:**

```
System.out.println ("Your change is " + count +
   (count == 1) ? "Dime" : "Dimes");
```

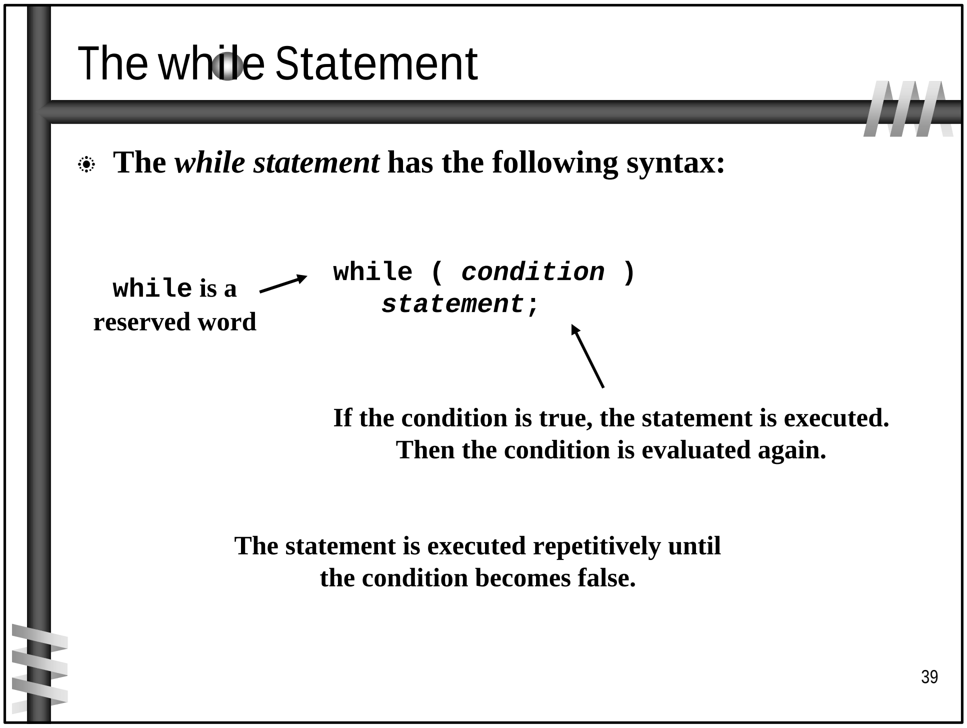
- **If `count` equals 1, then `"Dime"` is printed**
- **If `count` is anything other than 1, then `"Dimes"` is printed**

37



Repetition Statements

- ***Repetition statements* allow us to execute a statement multiple times repetitively**
- **They are often simply referred to as *loops***
- **Like conditional statements, they are controlled by boolean expressions**
- **Java has three kinds of repetition statements: the *while loop,* the *do loop,* and the *for loop***
- **The programmer must choose the right kind of loop for the situation**



The while Statement

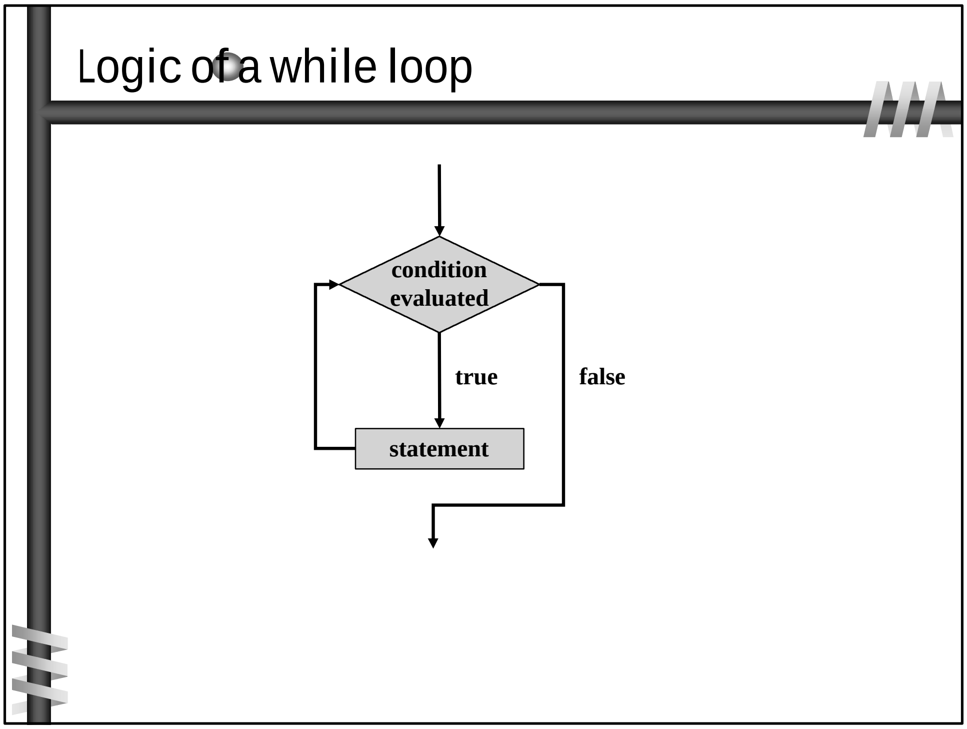
- The *while statement* has the following syntax:

```
while ( condition )
   statement;
```

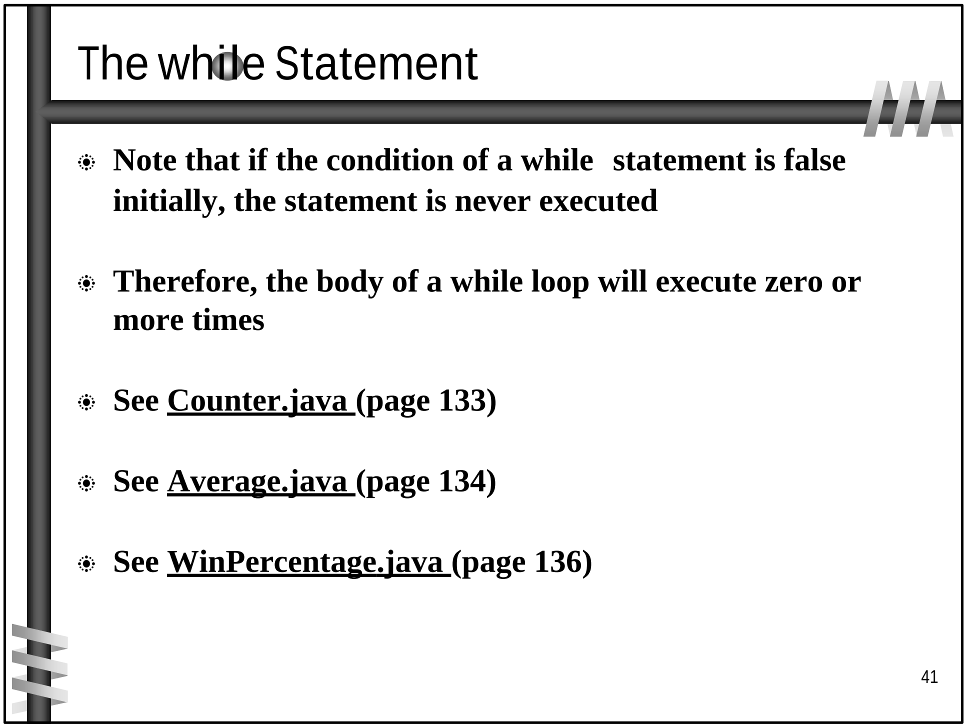
while is a reserved word

If the condition is true, the statement is executed. Then the condition is evaluated again.

The statement is executed repetitively until the condition becomes false.

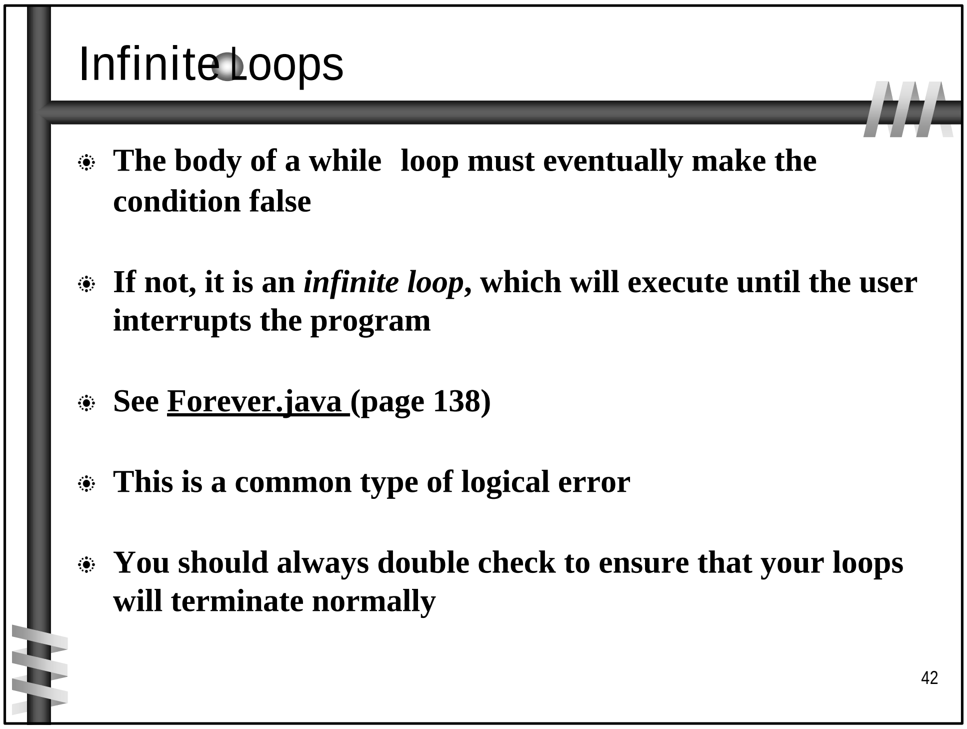


Logic of a while loop



The while Statement

- Note that if the condition of a while statement is false initially, the statement is never executed
- Therefore, the body of a while loop will execute zero or more times
- See <u>Counter.java</u> (page 133)
- See <u>Average.java</u> (page 134)
- See <u>WinPercentage.java</u> (page 136)



Infinite Loops

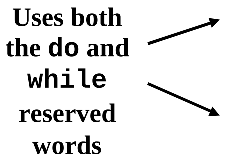
- The body of a while loop must eventually make the condition false
- If not, it is an *infinite loop*, which will execute until the user interrupts the program
- See <u>Forever.java</u> (page 138)
- This is a common type of logical error
- You should always double check to ensure that your loops will terminate normally

Nested Loops

- **Similar to nested if statements, loops can be nested as well**
- **That is, the body of a loop could contain another loop**
- **Each time through the outer loop, the inner loop will go through its entire set of iterations**
- **See PalindromeTester.java (page 137)**

The do Statement

- **The *do statement* has the following syntax:**



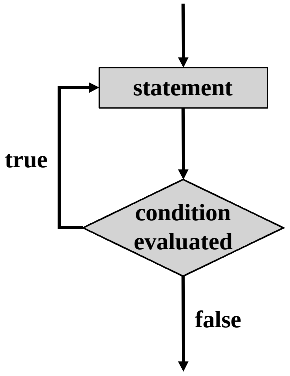
Uses both the `do` and `while` reserved words

```
do
{
   statement;
}
while ( condition )
```

The statement is executed once initially, then the condition is evaluated

The statement is repetitively executed until the condition becomes false

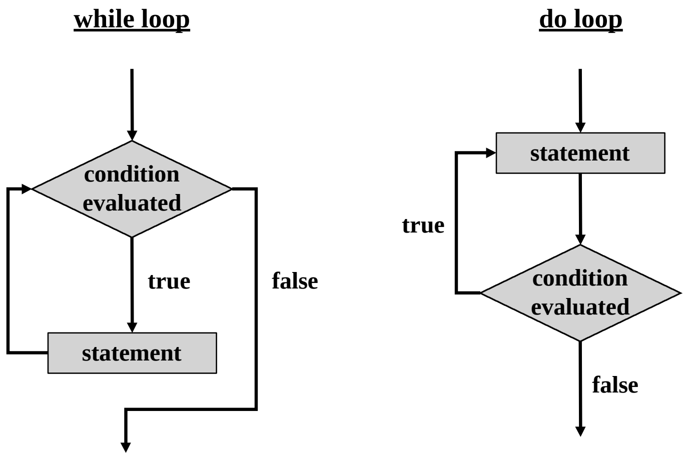
Logic of a do loop



The do Statement

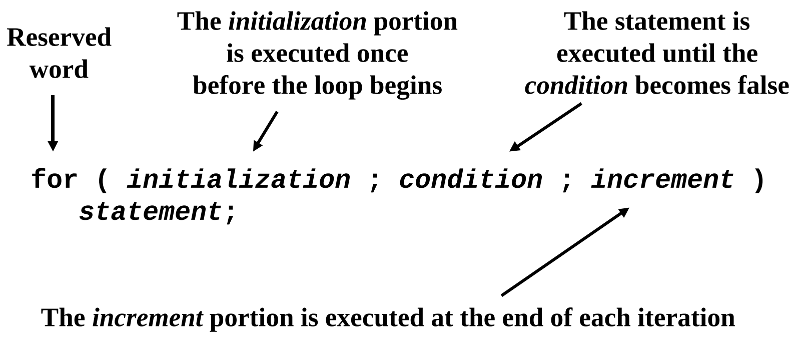
- A do loop is similar to a while loop, except that the condition is evaluated after the body of the loop is executed
- Therefore the body of a do loop will execute at least one time
- See Counter2.java (page 143)
- See ReverseNumber.java (page 144)

Comparing the while and do loops



The for Statement

- The *for statement* has the following syntax:

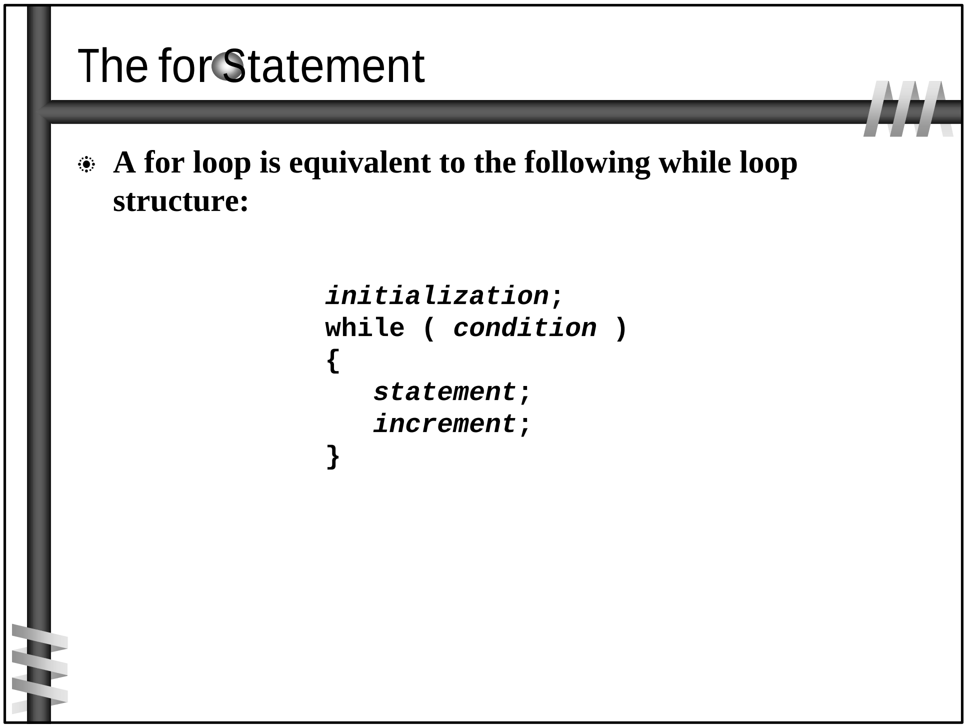

Reserved word

The *initialization* portion is executed once before the loop begins

The statement is executed until the *condition* becomes false

```
for ( initialization ; condition ; increment )
   statement;
```

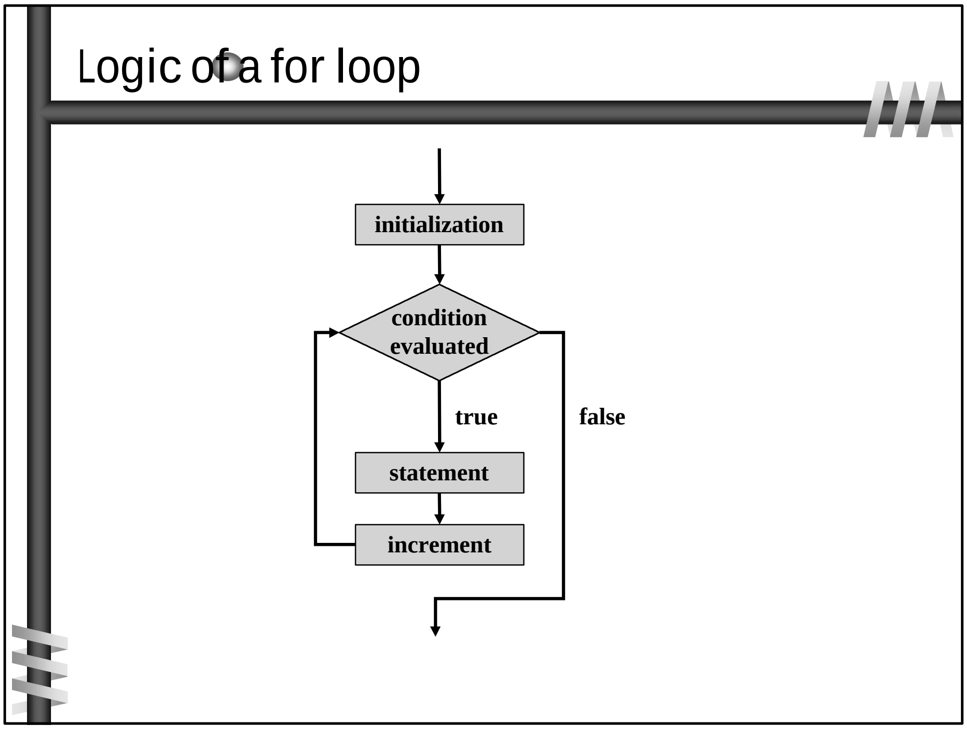
The *increment* portion is executed at the end of each iteration



The for Statement

- A for loop is equivalent to the following while loop structure:

```
initialization;
while ( condition )
{
   statement;
   increment;
}
```



Logic of a for loop

initialization

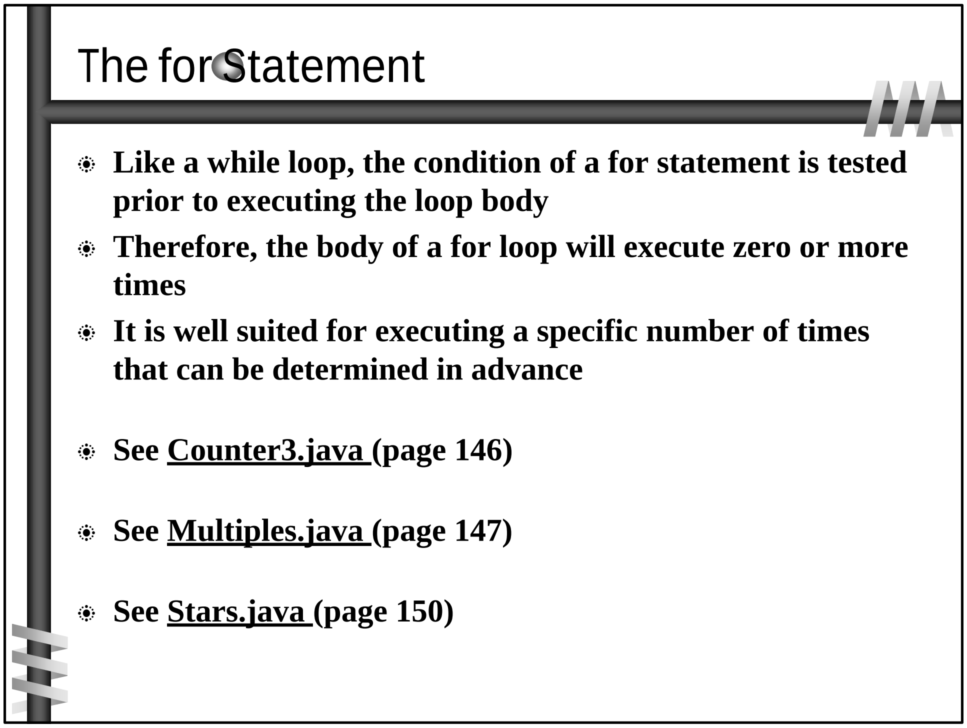
condition evaluated

true

false

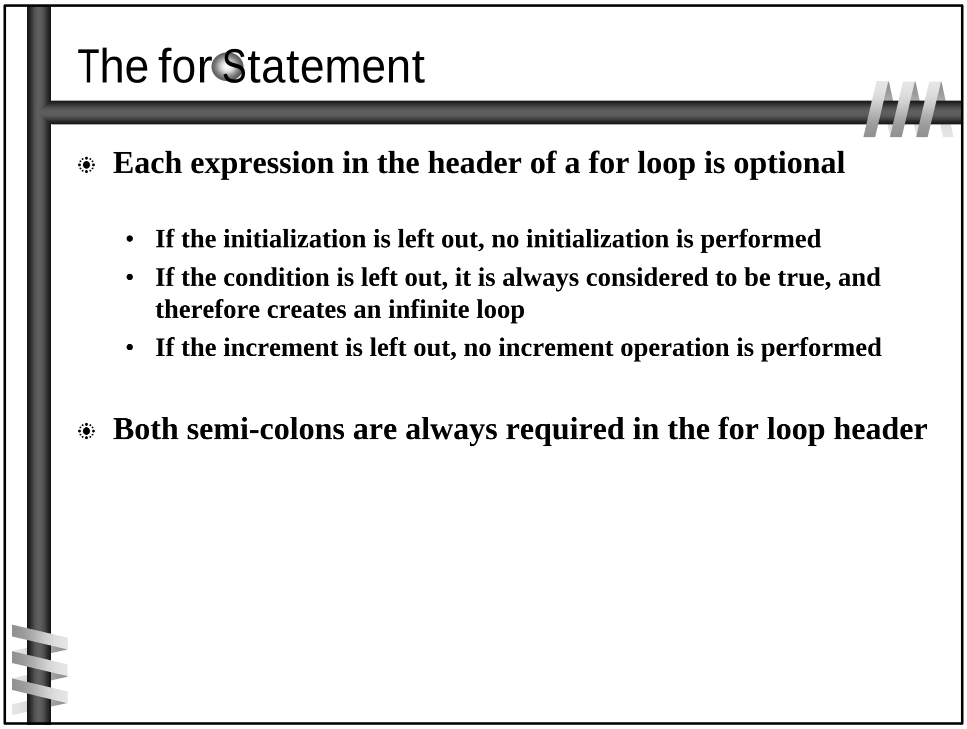
statement

increment



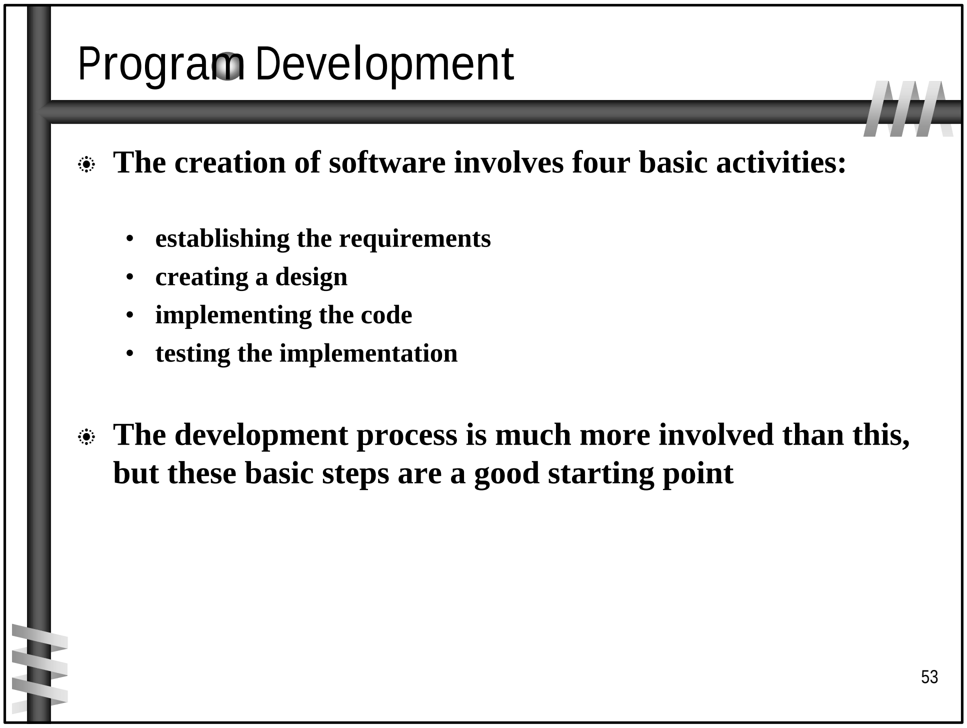
The for Statement

- Like a while loop, the condition of a for statement is tested prior to executing the loop body
- Therefore, the body of a for loop will execute zero or more times
- It is well suited for executing a specific number of times that can be determined in advance
- See Counter3.java (page 146)
- See Multiples.java (page 147)
- See Stars.java (page 150)



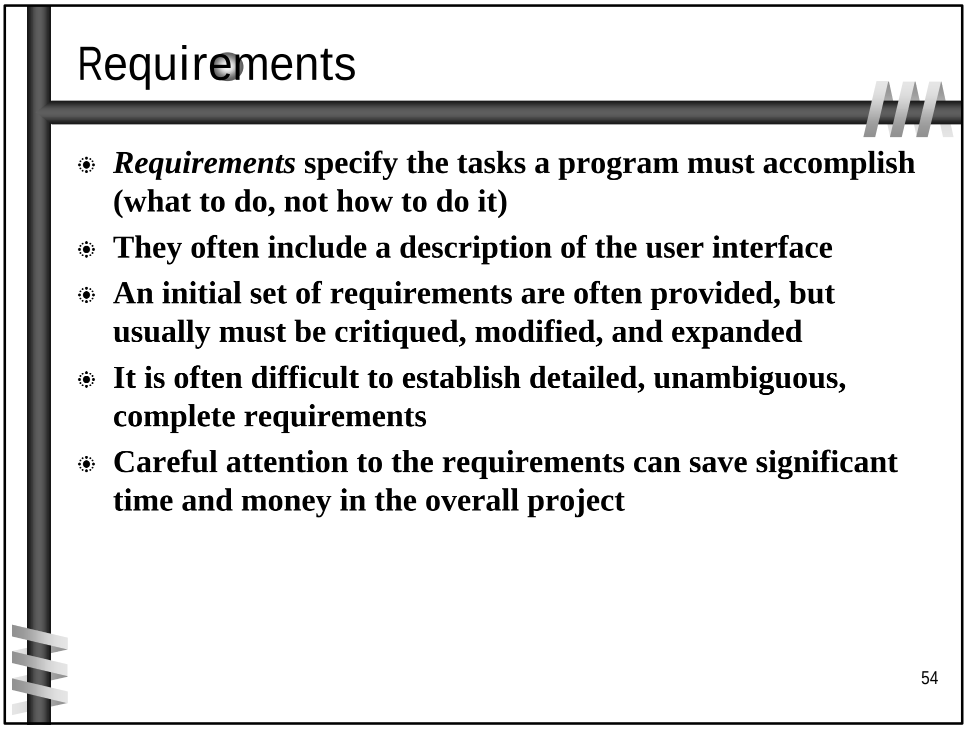
The for Statement

- Each expression in the header of a for loop is optional
  - If the initialization is left out, no initialization is performed
  - If the condition is left out, it is always considered to be true, and therefore creates an infinite loop
  - If the increment is left out, no increment operation is performed
- Both semi-colons are always required in the for loop header



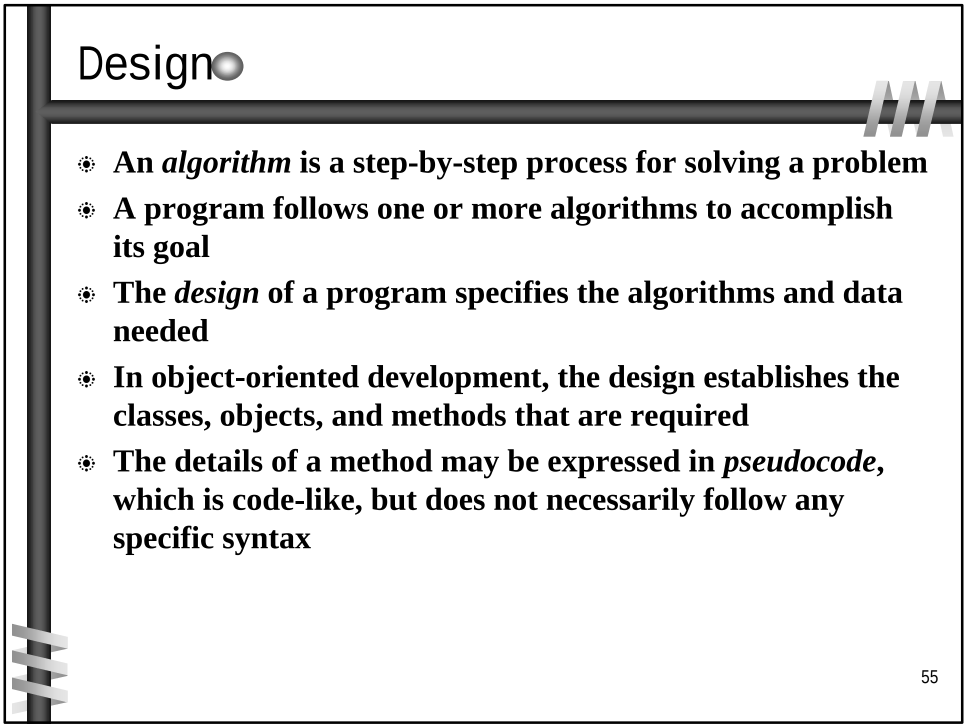
Program Development

- The creation of software involves four basic activities:
  - establishing the requirements
  - creating a design
  - implementing the code
  - testing the implementation
- The development process is much more involved than this, but these basic steps are a good starting point



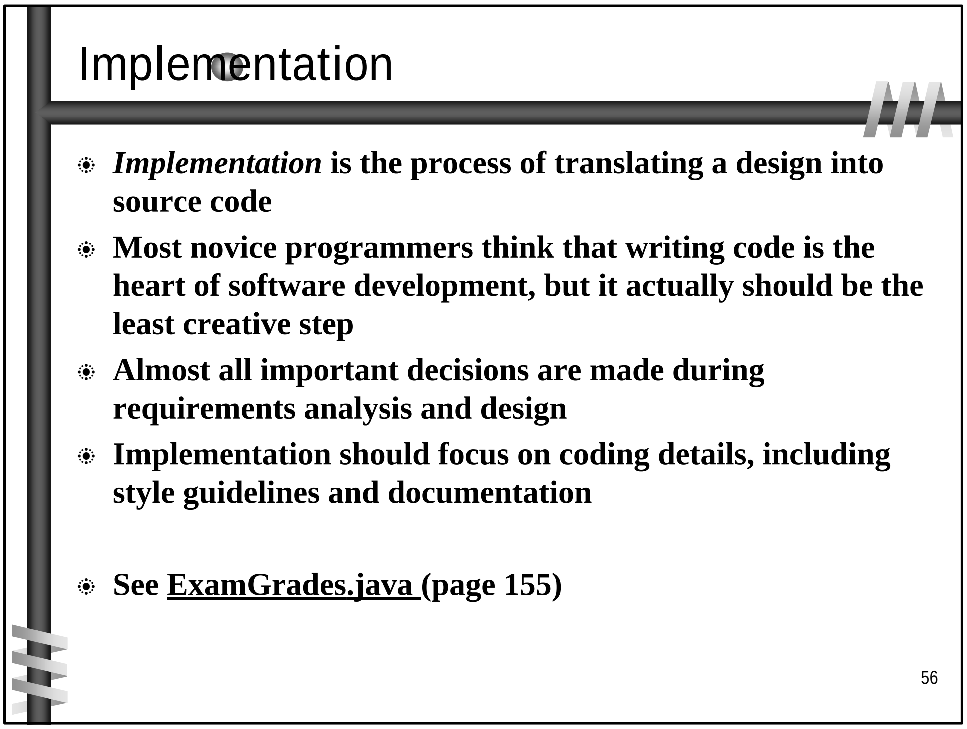
Requirements

- *Requirements* specify the tasks a program must accomplish (what to do, not how to do it)
- They often include a description of the user interface
- An initial set of requirements are often provided, but usually must be critiqued, modified, and expanded
- It is often difficult to establish detailed, unambiguous, complete requirements
- Careful attention to the requirements can save significant time and money in the overall project



Design

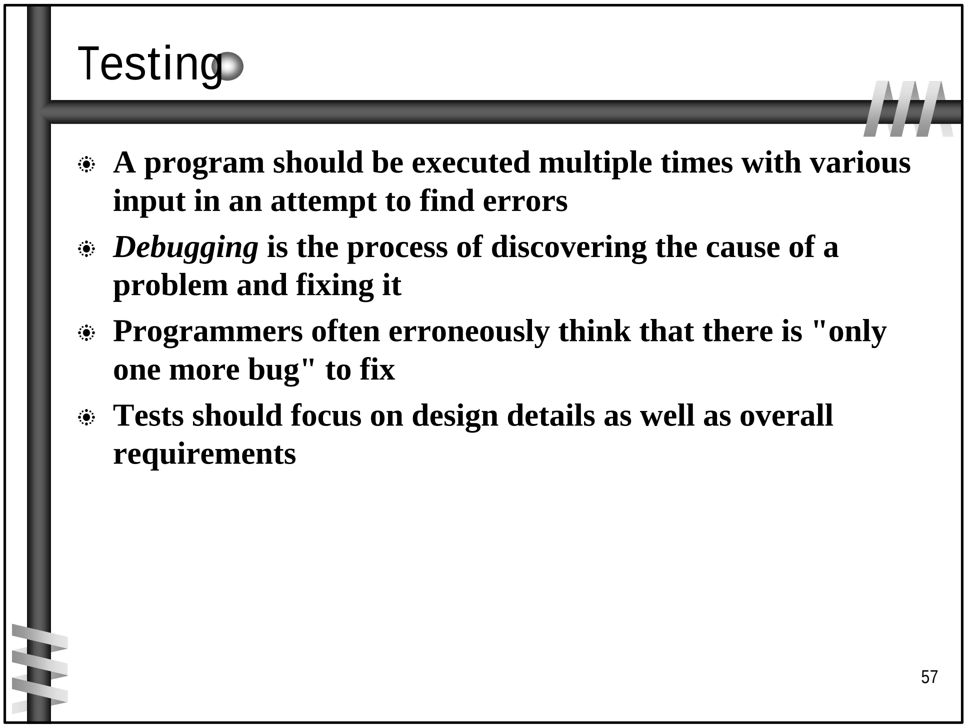
- An *algorithm* is a step-by-step process for solving a problem
- A program follows one or more algorithms to accomplish its goal
- The *design* of a program specifies the algorithms and data needed
- In object-oriented development, the design establishes the classes, objects, and methods that are required
- The details of a method may be expressed in *pseudocode*, which is code-like, but does not necessarily follow any specific syntax



Implementation

- *Implementation* is the process of translating a design into source code
- Most novice programmers think that writing code is the heart of software development, but it actually should be the least creative step
- Almost all important decisions are made during requirements analysis and design
- Implementation should focus on coding details, including style guidelines and documentation

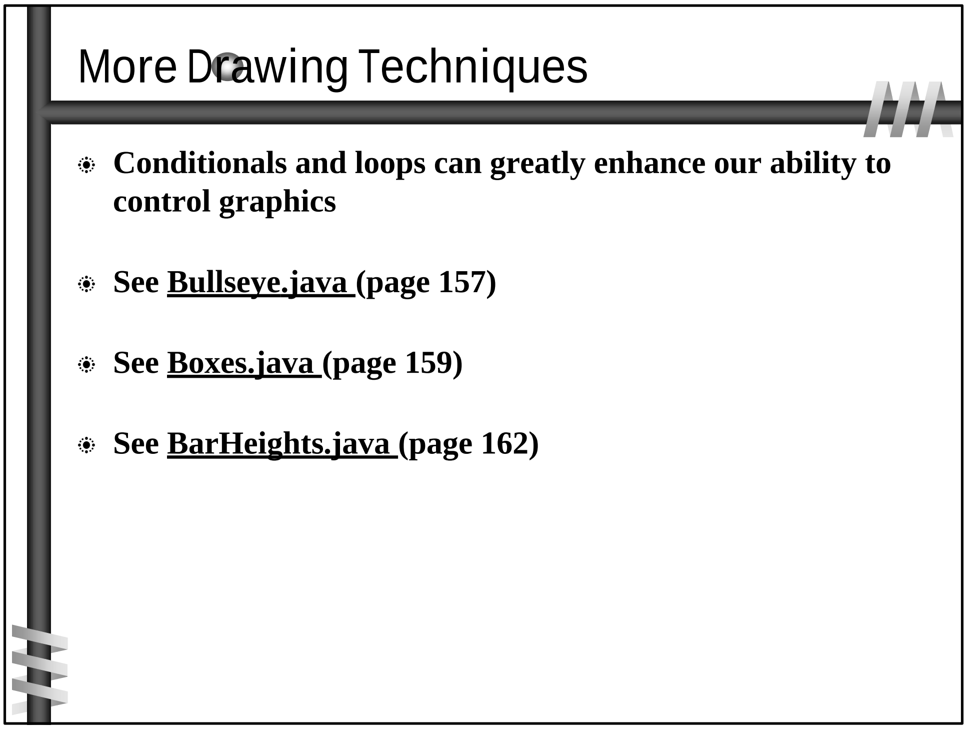
- See ExamGrades.java (page 155)



Testing

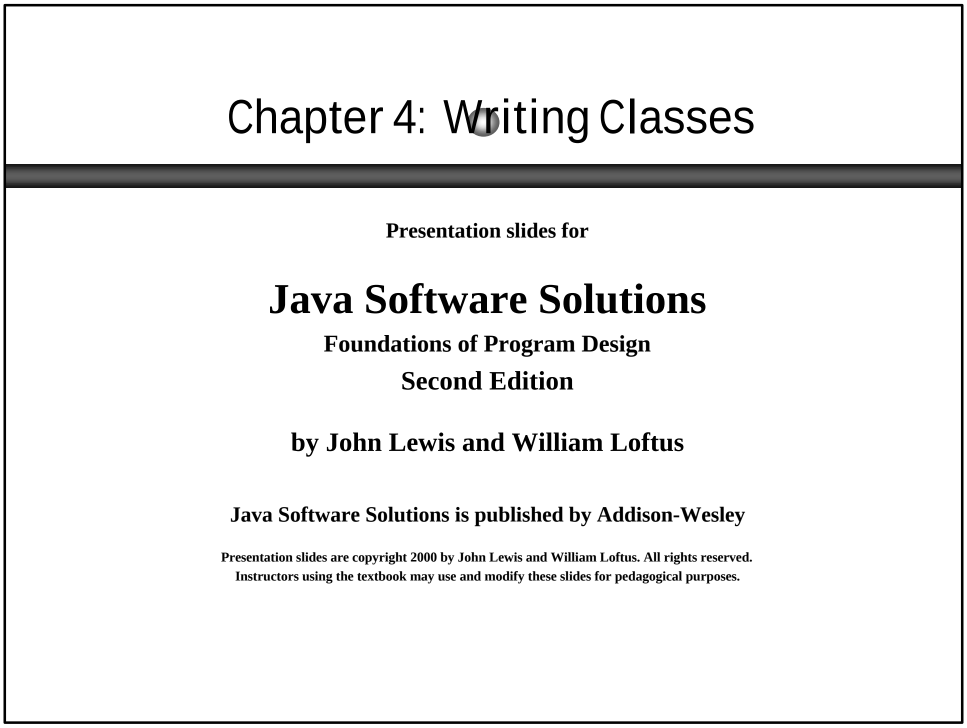
- A program should be executed multiple times with various input in an attempt to find errors
- *Debugging* is the process of discovering the cause of a problem and fixing it
- Programmers often erroneously think that there is "only one more bug" to fix
- Tests should focus on design details as well as overall requirements

57



More Drawing Techniques

- Conditionals and loops can greatly enhance our ability to control graphics
- See Bullseye.java (page 157)
- See Boxes.java (page 159)
- See BarHeights.java (page 162)



Chapter 4: Writing Classes

Presentation slides for

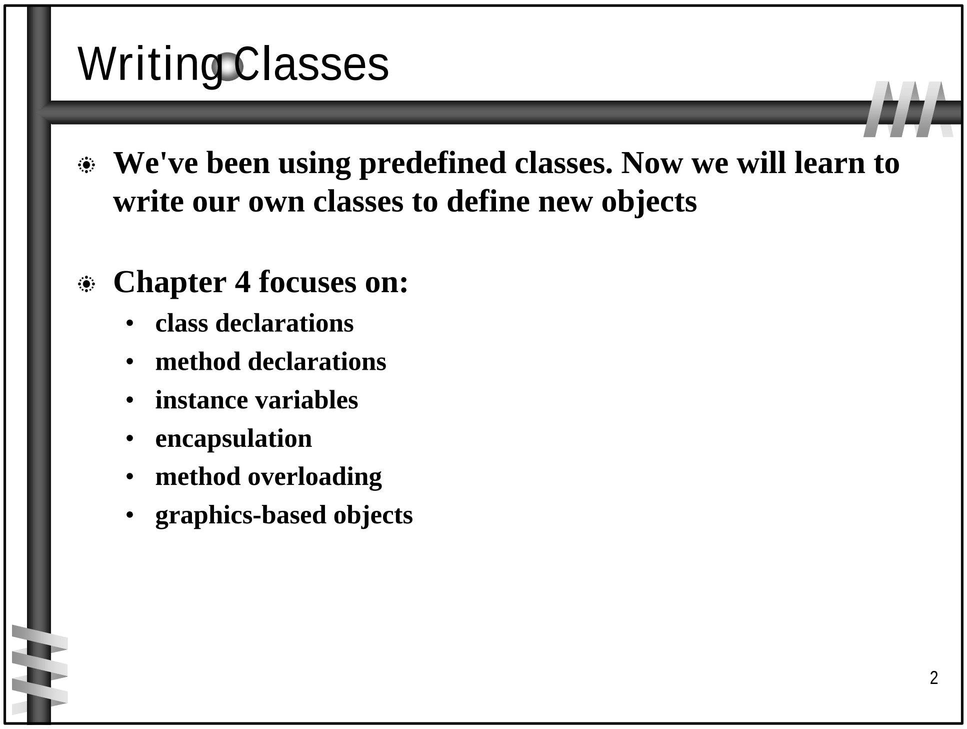
Java Software Solutions

Foundations of Program Design

Second Edition

by John Lewis and William Loftus

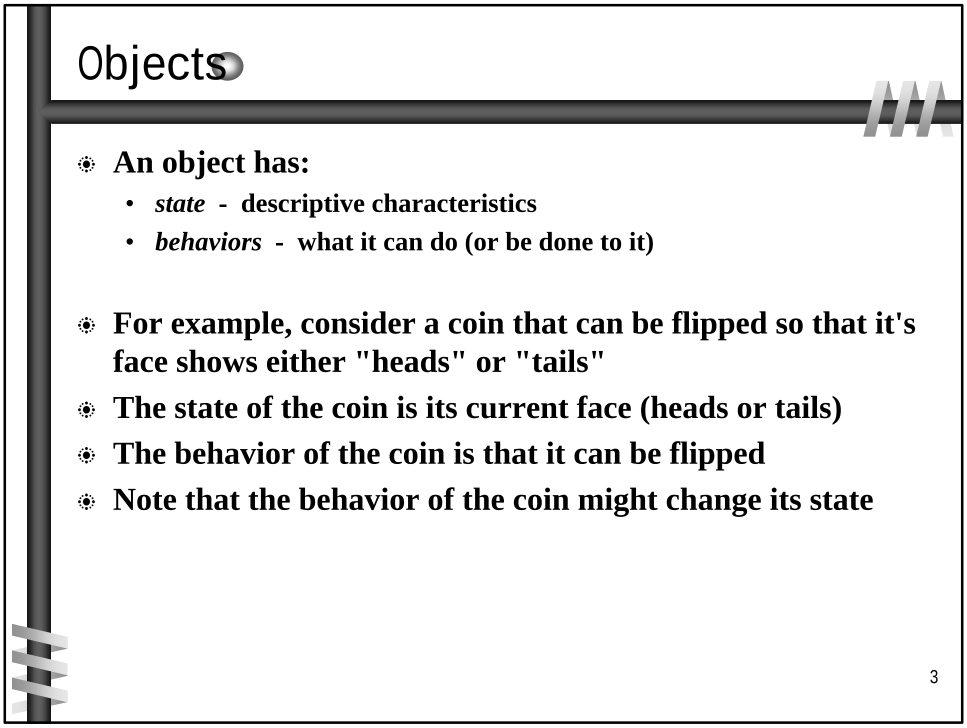
Java Software Solutions is published by Addison-Wesley



Writing Classes

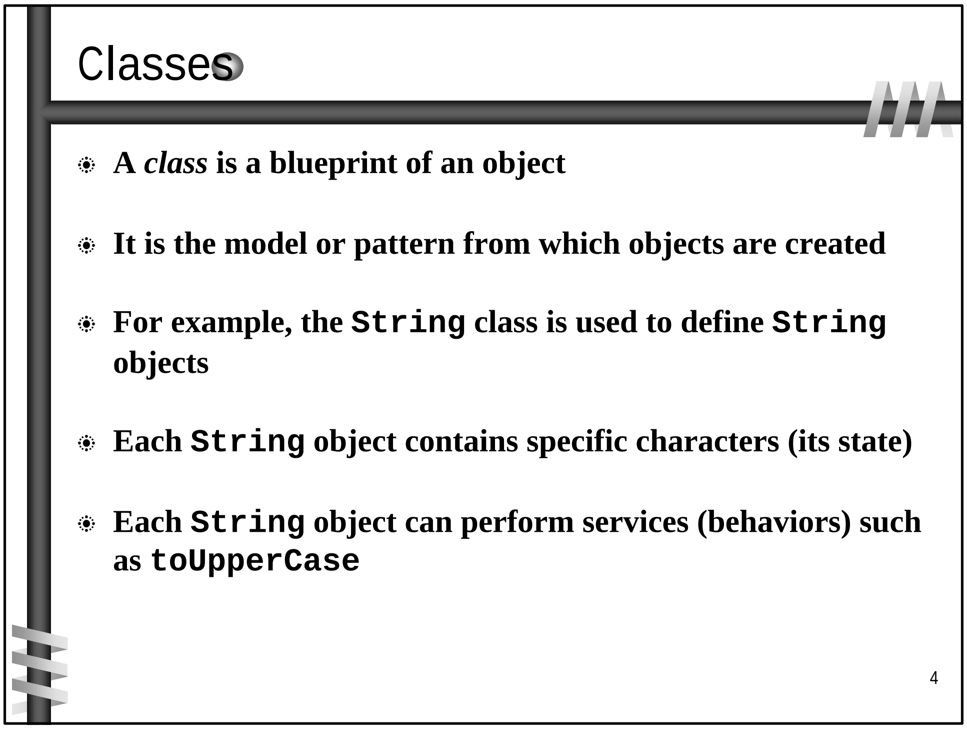
- We've been using predefined classes. Now we will learn to write our own classes to define new objects

- Chapter 4 focuses on:
  - class declarations
  - method declarations
  - instance variables
  - encapsulation
  - method overloading
  - graphics-based objects



Objects

- An object has:
  - *state* - descriptive characteristics
  - *behaviors* - what it can do (or be done to it)
- For example, consider a coin that can be flipped so that it's face shows either "heads" or "tails"
- The state of the coin is its current face (heads or tails)
- The behavior of the coin is that it can be flipped
- Note that the behavior of the coin might change its state



Classes

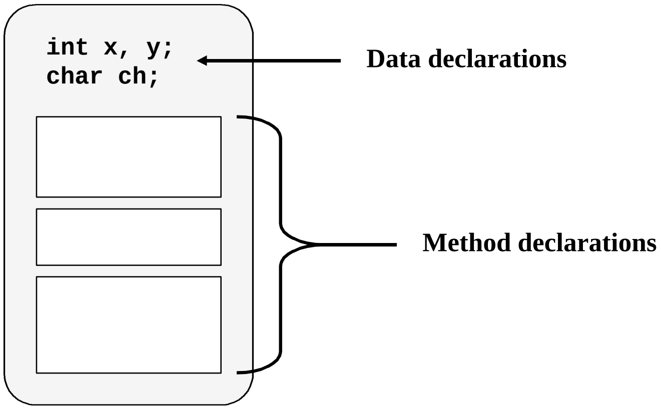
- A *class* is a blueprint of an object
- It is the model or pattern from which objects are created
- For example, the `String` class is used to define `String` objects
- Each `String` object contains specific characters (its state)
- Each `String` object can perform services (behaviors) such as `toUpperCase`

Classes

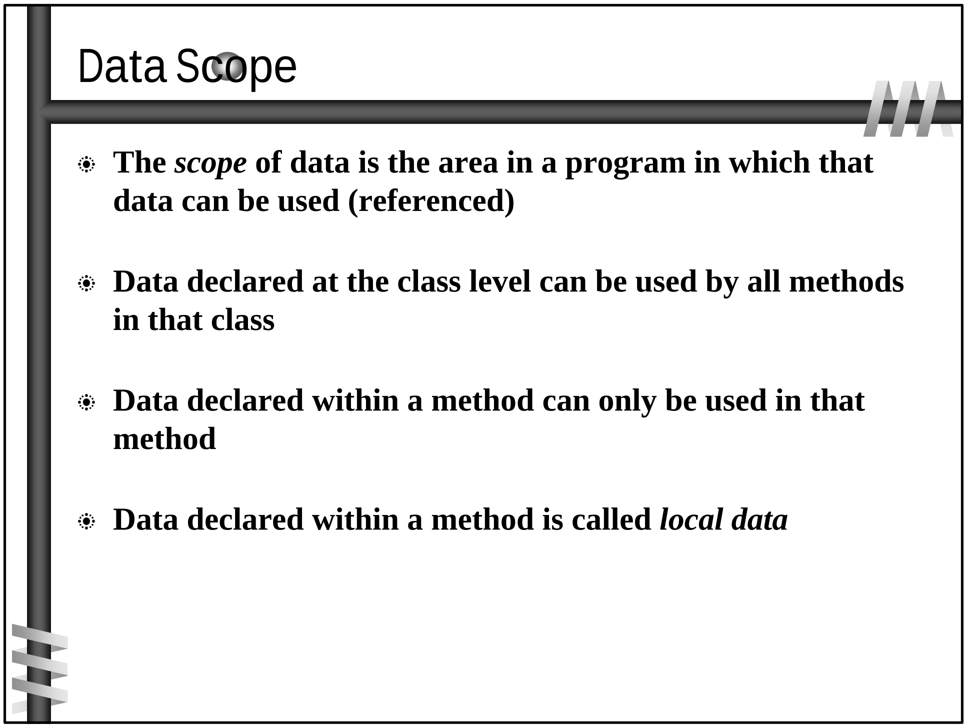
- The `String` class was provided for us by the Java standard class library
- But we can also write our own classes that define specific objects that we need
- For example, suppose we wanted to write a program that simulates the flipping of a coin
- We could write a `Coin` class to represent a coin object

Classes

- A class contains data declarations and method declarations

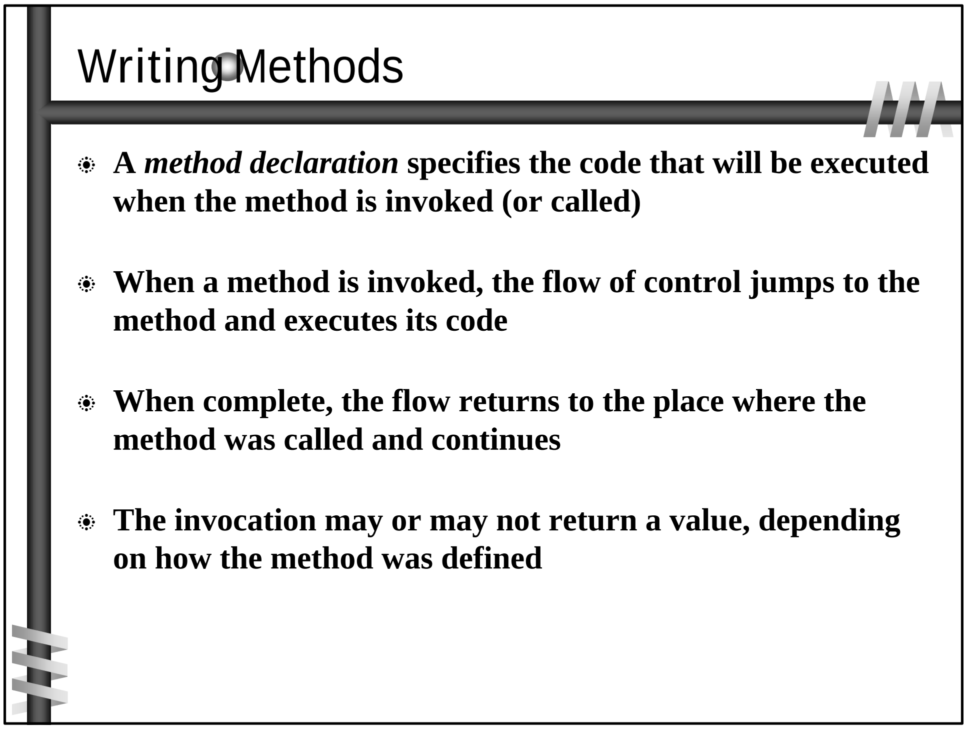


```
int x, y;
char ch;
```



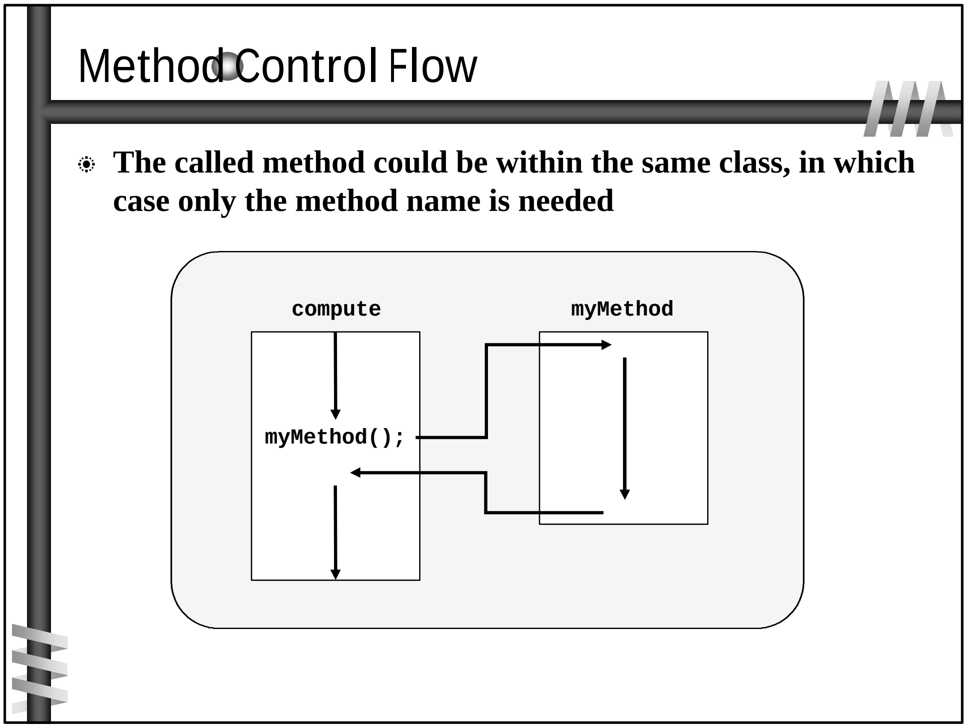
Data Scope

- The *scope* of data is the area in a program in which that data can be used (referenced)
- Data declared at the class level can be used by all methods in that class
- Data declared within a method can only be used in that method
- Data declared within a method is called *local data*



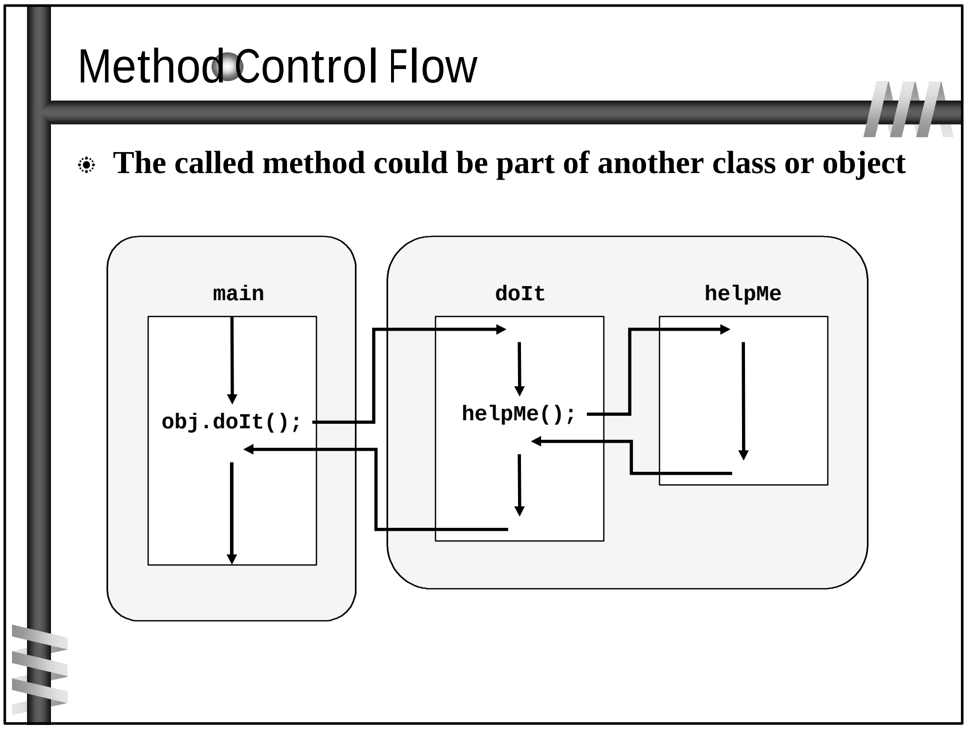
Writing Methods

- A *method declaration* specifies the code that will be executed when the method is invoked (or called)
- When a method is invoked, the flow of control jumps to the method and executes its code
- When complete, the flow returns to the place where the method was called and continues
- The invocation may or may not return a value, depending on how the method was defined



Method Control Flow

- The called method could be within the same class, in which case only the method name is needed



Method Control Flow

- The called method could be part of another class or object

The Coin Class

- In our **`Coin`** class we could define the following data:
  - **`face`, an integer that represents the current face**
  - **`HEADS` and `TAILS`, integer constants that represent the two possible states**
- **We might also define the following methods:**
  - **a `Coin` constructor, to set up the object**
  - **a `flip` method, to flip the coin**
  - **a `getFace` method, to return the current face**
  - **a `toString` method, to return a string description for printing**

The Coin Class

- **See <u>CountFlips.java</u> (page 179)**
- **See <u>Coin.java</u> (page 180)**
- **Once the `Coin` class has been defined, we can use it again in other programs as needed**
- **Note that the `CountFlips` program did not use the `toString` method**
- **A program will not necessarily use every service provided by an object**

Instance Data

- The `face` variable in the `Coin` class is called *instance data* because each instance (object) of the `Coin` class has its own

- A class declares the type of the data, but it does not reserve any memory space for it

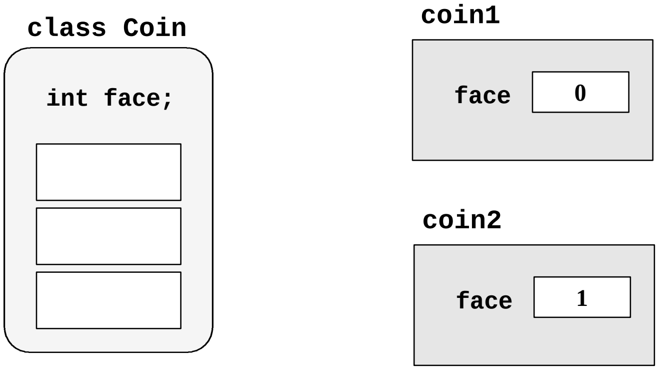
- Every time a `Coin` object is created, a new `face` variable is created as well

- The objects of a class share the method definitions, but they have unique data space

- That's the only way two objects can have different states

Instance Data

- See FlipRace.java (page 182)



class Coin

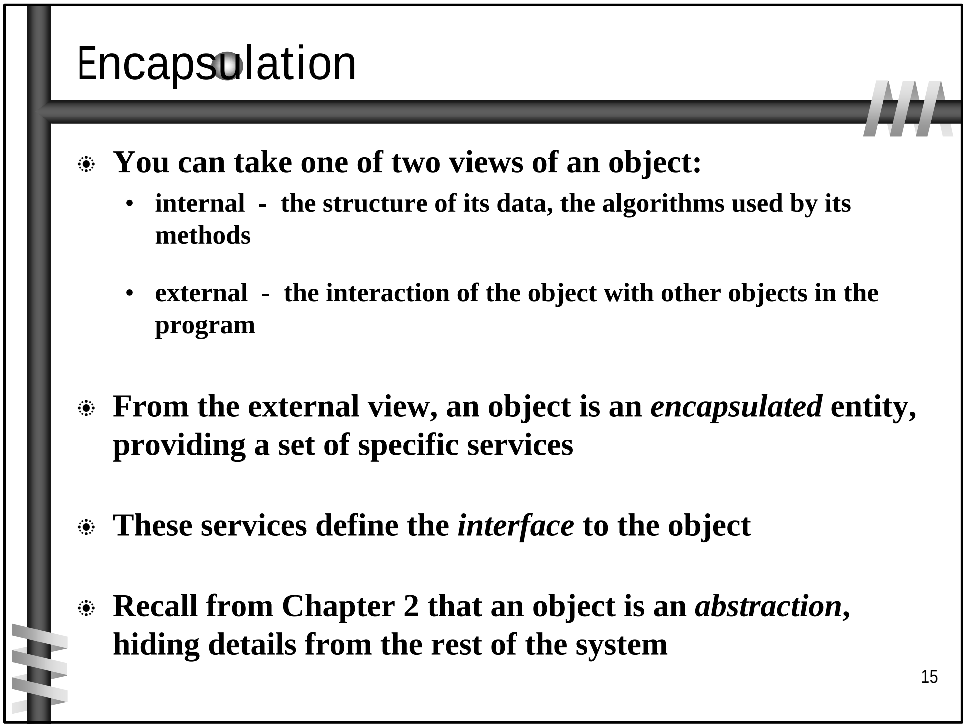
int face;

coin1

face 0

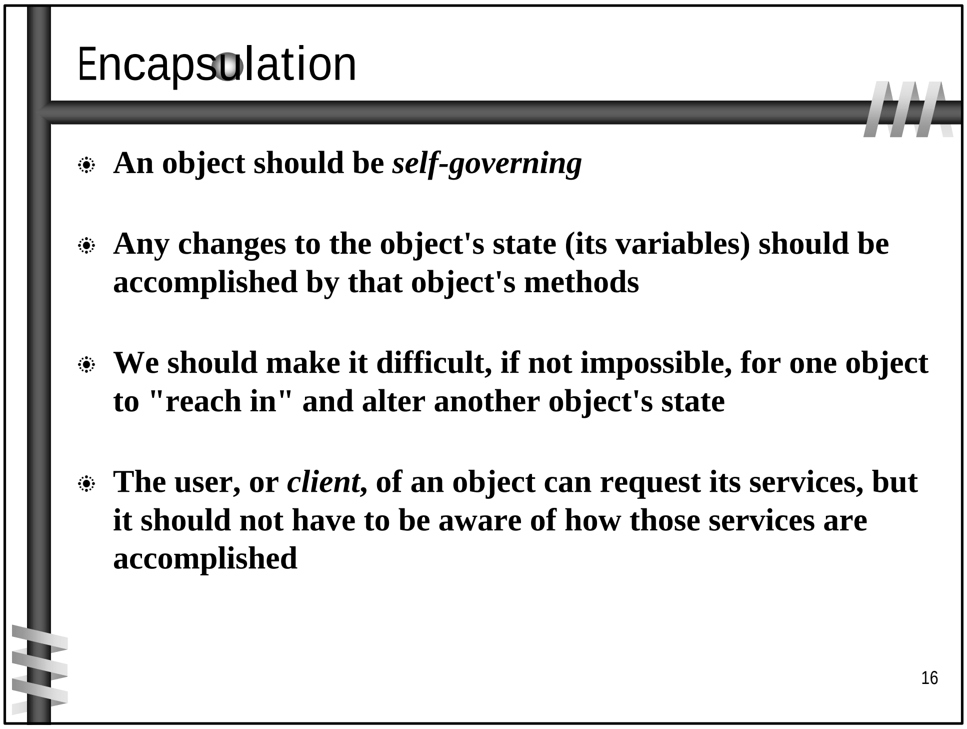
coin2

face 1



Encapsulation

- **You can take one of two views of an object:**
  - **internal - the structure of its data, the algorithms used by its methods**
  - **external - the interaction of the object with other objects in the program**
- **From the external view, an object is an *encapsulated* entity, providing a set of specific services**
- **These services define the *interface* to the object**
- **Recall from Chapter 2 that an object is an *abstraction*, hiding details from the rest of the system**

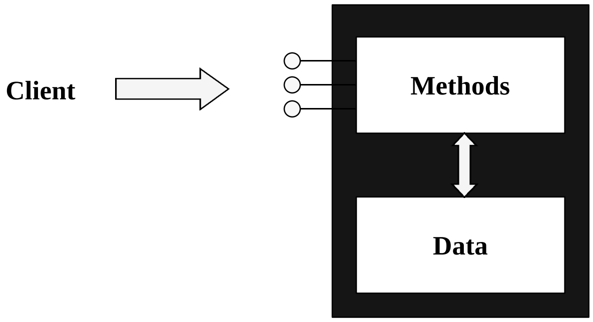


Encapsulation

- **An object should be *self-governing***
- **Any changes to the object's state (its variables) should be accomplished by that object's methods**
- **We should make it difficult, if not impossible, for one object to "reach in" and alter another object's state**
- **The user, or *client*, of an object can request its services, but it should not have to be aware of how those services are accomplished**

Encapsulation

- An encapsulated object can be thought of as a *black box*
- Its inner workings are hidden to the client, which only invokes the interface methods



Visibility Modifiers

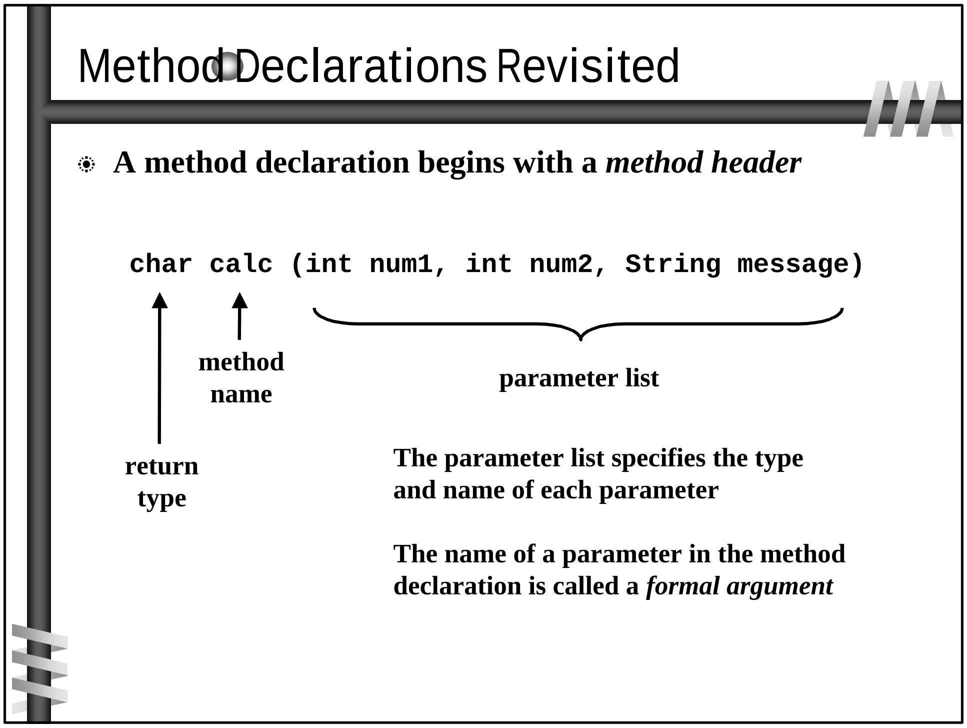
- In Java, we accomplish encapsulation through the appropriate use of *visibility modifiers*
- A *modifier* is a Java reserved word that specifies particular characteristics of a method or data value
- We've used the modifier `final` to define a constant
- Java has three visibility modifiers: `public`, `private`, and `protected`
- We will discuss the `protected` modifier later

Visibility Modifiers

- **Members of a class that are declared with *public visibility* can be accessed from anywhere**
- **Members of a class that are declared with *private visibility* can only be accessed from inside the class**
- **Members declared without a visibility modifier have *default visibility* and can be accessed by any class in the same package**
- **Java modifiers are discussed in detail in Appendix F**

Visibility Modifiers

- **As a general rule, no object's data should be declared with public visibility**
- **Methods that provide the object's services are usually declared with public visibility so that they can be invoked by clients**
- **Public methods are also called *service methods***
- **A method created simply to assist a service method is called a *support method***
- **Since a support method is not intended to be called by a client, it should not be declared with public visibility**



Method Declarations Revisited

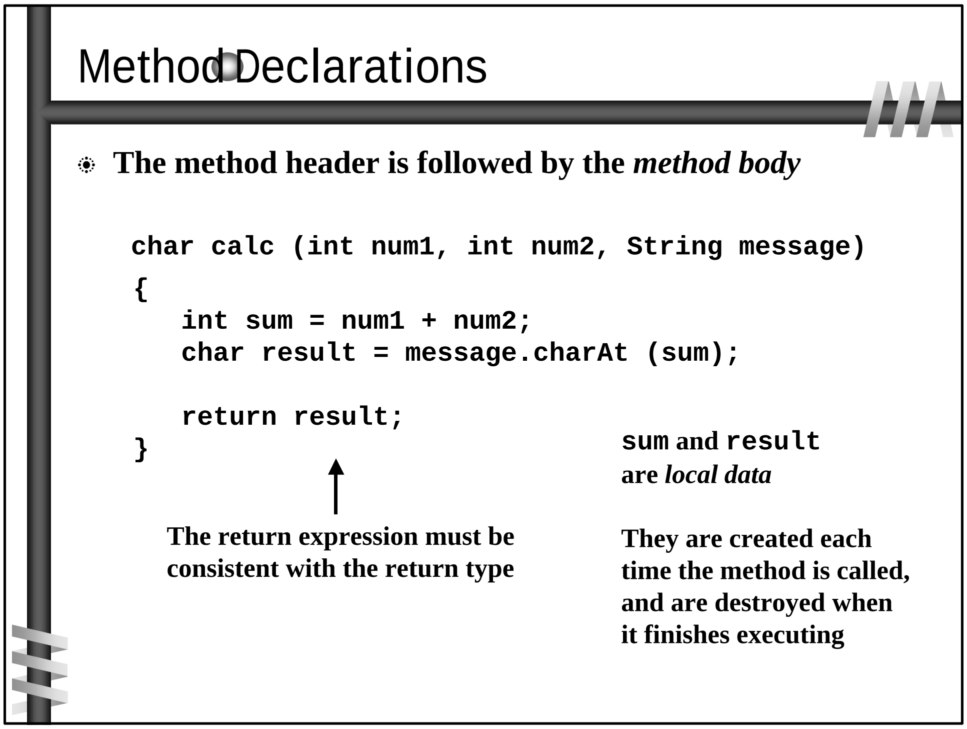
- A method declaration begins with a *method header*

```
char calc (int num1, int num2, String message)
```

return type
method name
parameter list

The parameter list specifies the type and name of each parameter

The name of a parameter in the method declaration is called a *formal argument*



Method Declarations

- The method header is followed by the *method body*

```
char calc (int num1, int num2, String message)
{
   int sum = num1 + num2;
   char result = message.charAt (sum);

   return result;
}
```

The return expression must be consistent with the return type

`sum` and `result` are *local data*

They are created each time the method is called, and are destroyed when it finishes executing

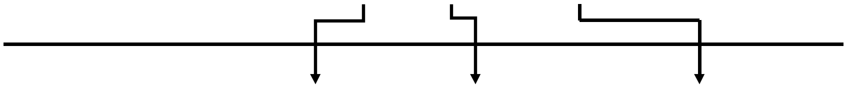
The return Statement

- The *return type* of a method indicates the type of value that the method sends back to the calling location
- A method that does not return a value has a `void` return type
- The *return statement* specifies the value that will be returned
- Its expression must conform to the return type

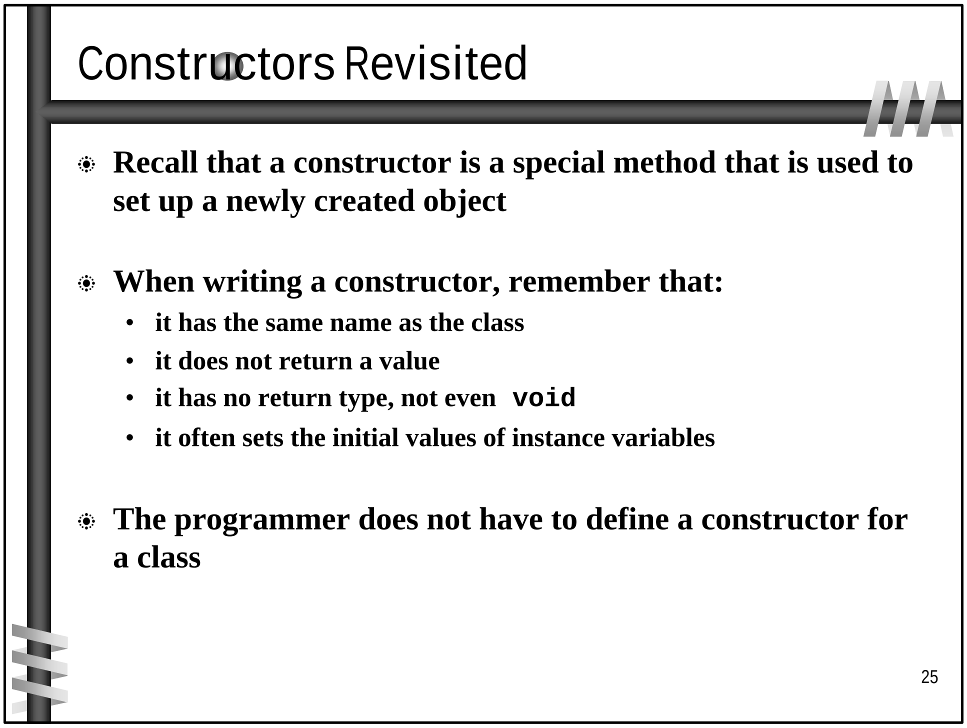
Parameters

- Each time a method is called, the *actual arguments* in the invocation are copied into the formal arguments

```
     ch = obj.calc (25, count, "Hello");
```



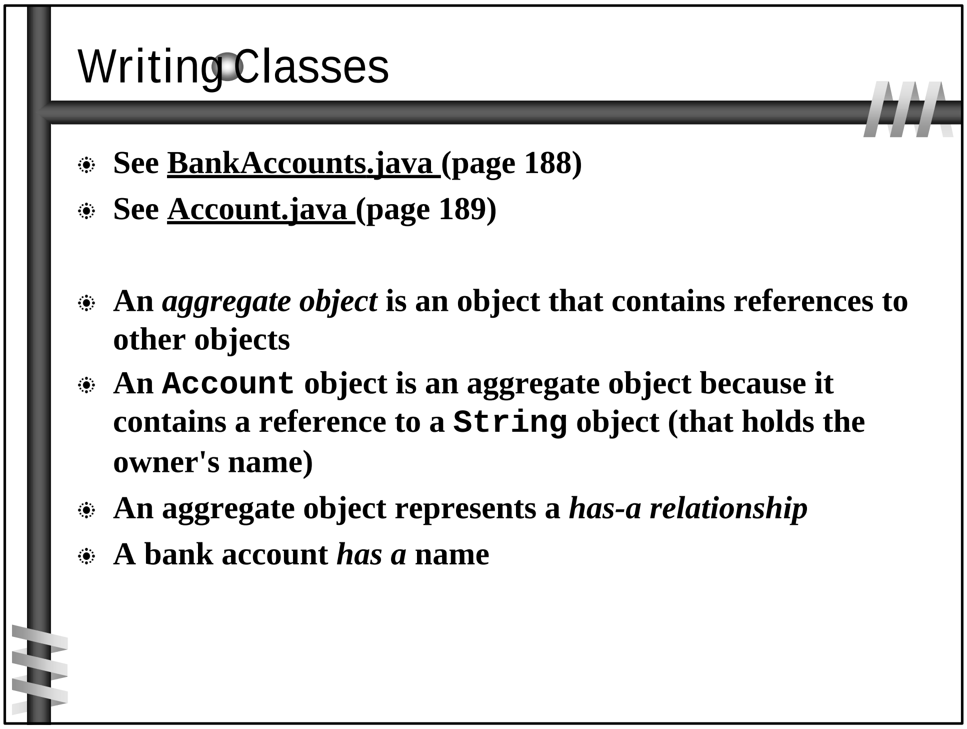
```
char calc (int num1, int num2, String message)
{
   int sum = num1 + num2;
   char result = message.charAt (sum);

   return result;
}
```



Constructors Revisited

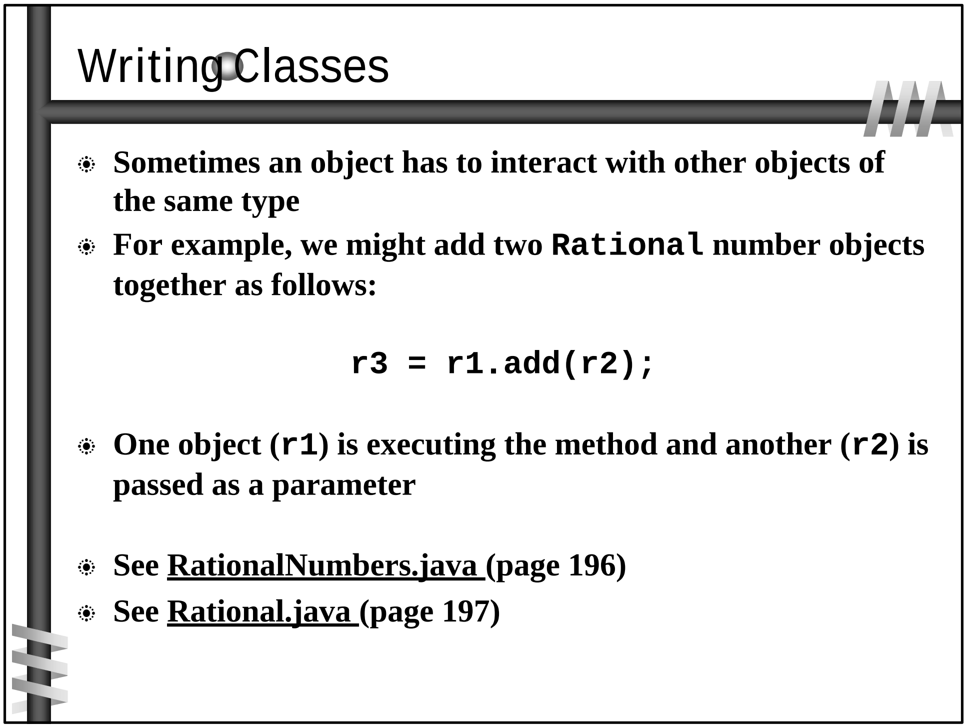
- Recall that a constructor is a special method that is used to set up a newly created object
- When writing a constructor, remember that:
  - it has the same name as the class
  - it does not return a value
  - it has no return type, not even `void`
  - it often sets the initial values of instance variables
- The programmer does not have to define a constructor for a class

25



Writing Classes

- See BankAccounts.java (page 188)
- See Account.java (page 189)
- An *aggregate object* is an object that contains references to other objects
- An `Account` object is an aggregate object because it contains a reference to a `String` object (that holds the owner's name)
- An aggregate object represents a *has-a relationship*
- A bank account *has a* name

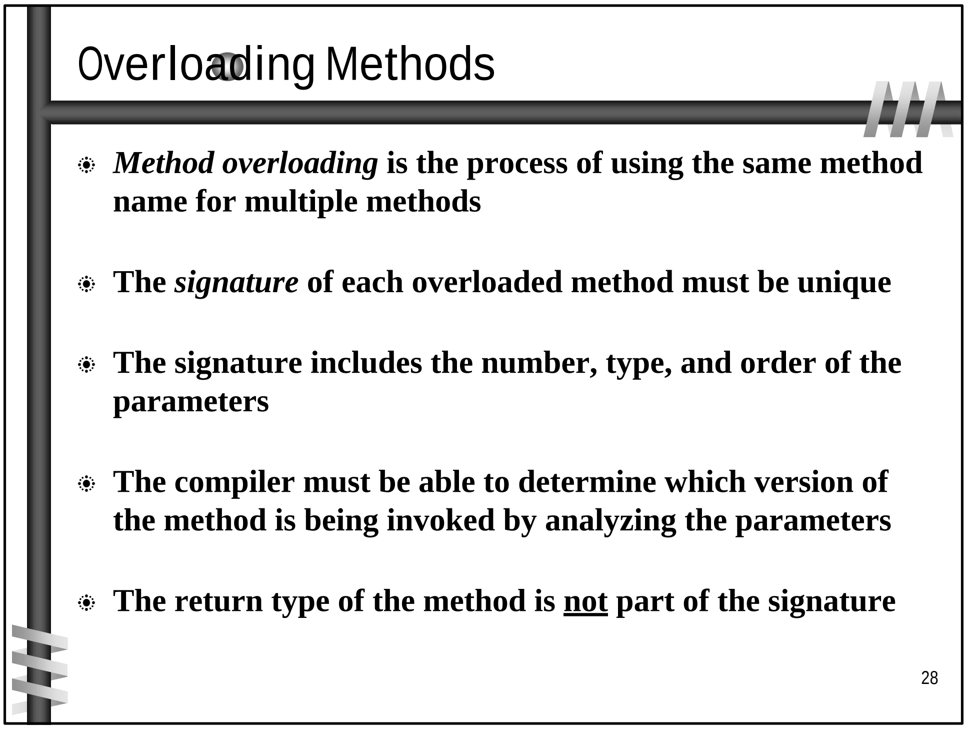


Writing Classes

- Sometimes an object has to interact with other objects of the same type
- For example, we might add two `Rational` number objects together as follows:

```
r3 = r1.add(r2);
```

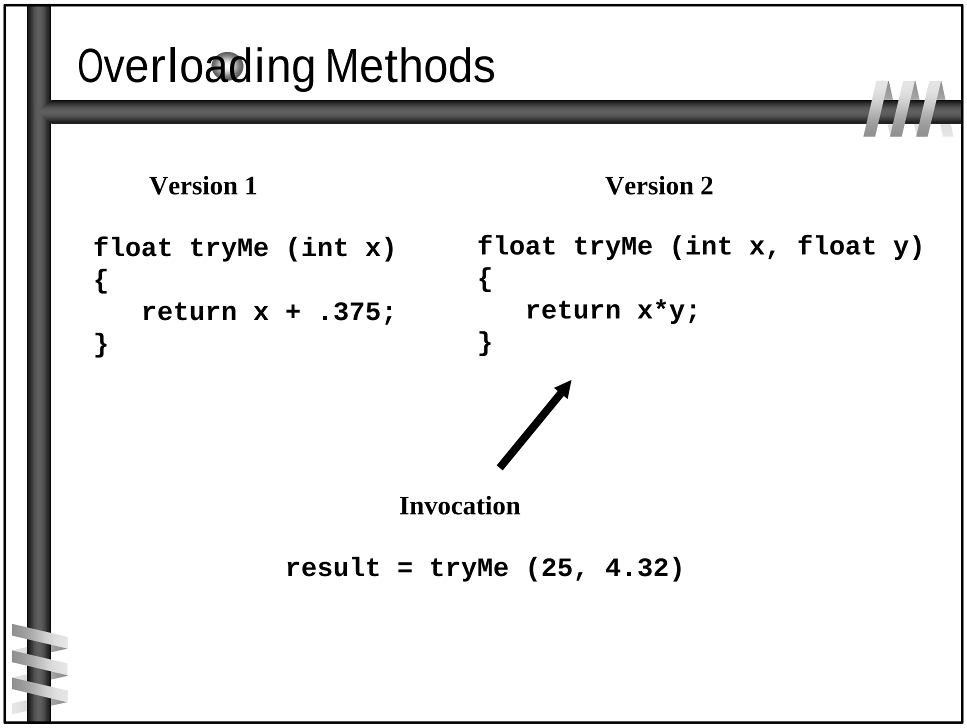
- One object (`r1`) is executing the method and another (`r2`) is passed as a parameter
- See <u>RationalNumbers.java</u> (page 196)
- See <u>Rational.java</u> (page 197)



Overloading Methods

- *Method overloading* is the process of using the same method name for multiple methods
- The *signature* of each overloaded method must be unique
- The signature includes the number, type, and order of the parameters
- The compiler must be able to determine which version of the method is being invoked by analyzing the parameters
- The return type of the method is <u>not</u> part of the signature

28



Overloading Methods

Version 1

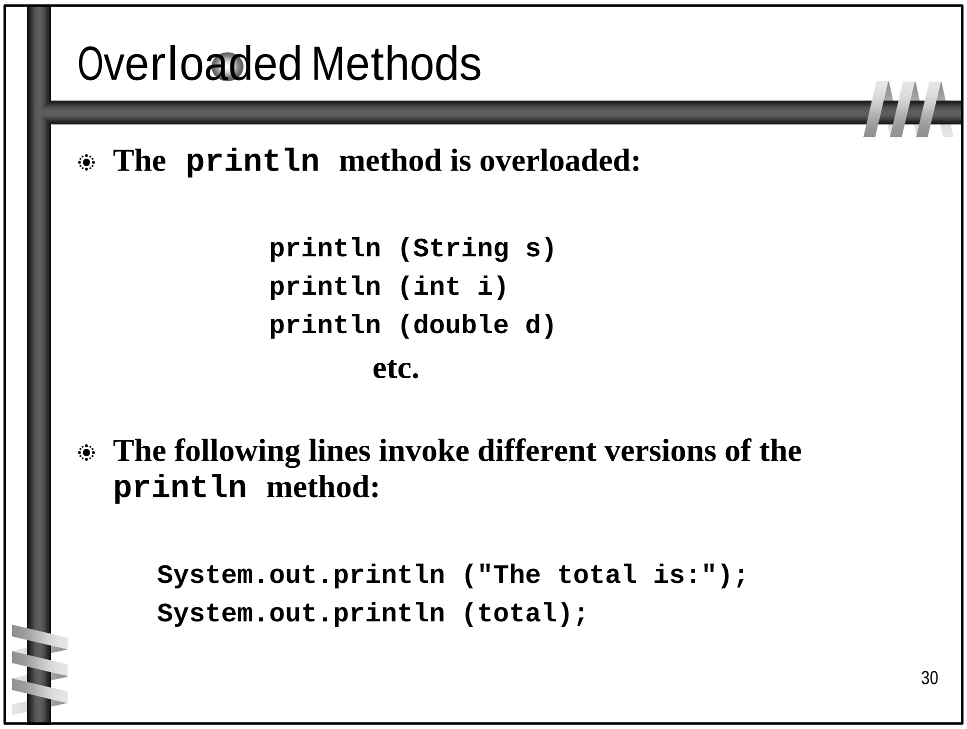
```
float tryMe (int x)
{
   return x + .375;
}
```

Version 2

```
float tryMe (int x, float y)
{
   return x*y;
}
```

Invocation

```
result = tryMe (25, 4.32)
```



Overloaded Methods

- **The `println` method is overloaded:**

```
println (String s)
println (int i)
println (double d)
```

etc.

- **The following lines invoke different versions of the `println` method:**

```
System.out.println ("The total is:");
System.out.println (total);
```

Overloading Methods

- **Constructors can be overloaded**
- **An overloaded constructor provides multiple ways to set up a new object**

- **See <u>SnakeEyes.java</u> (page 203)**
- **See <u>Die.java</u> (page 204)**

The StringTokenizer Class

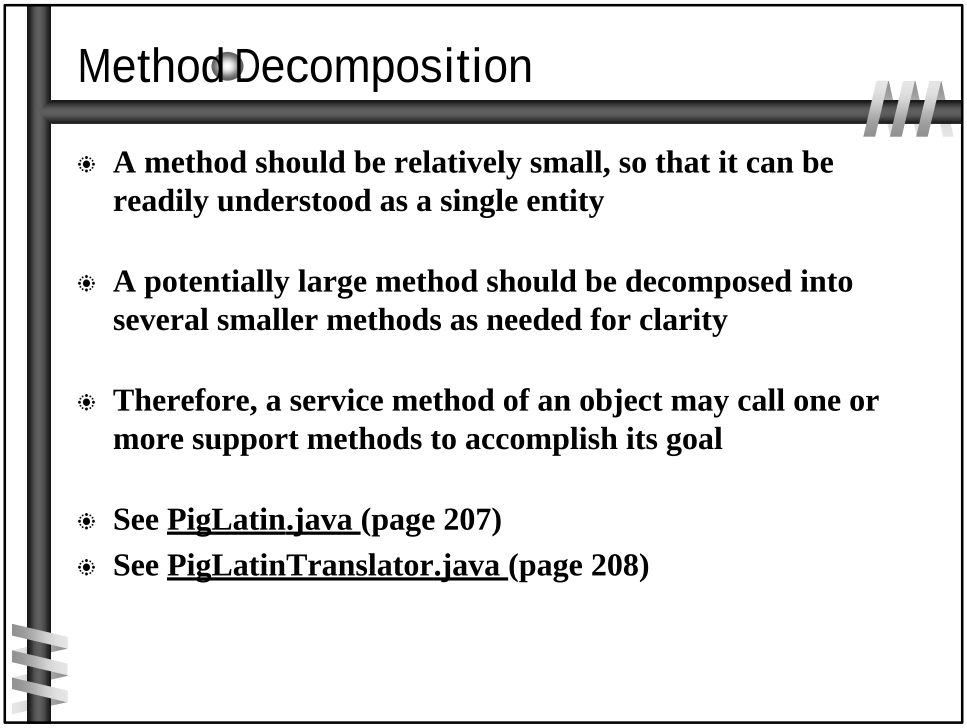
- **The next example makes use of the `StringTokenizer` class, which is defined in the `java.util` package**

- **A `StringTokenizer` object separates a string into smaller substrings (tokens)**

- **By default, the tokenizer separates the string at white space**

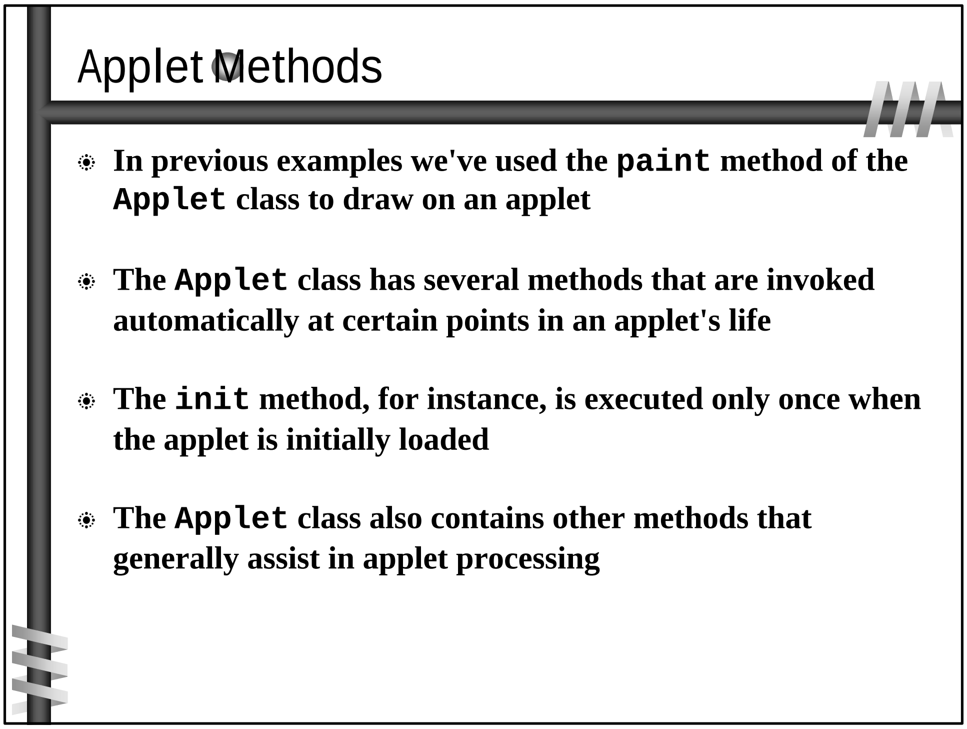
- **The `StringTokenizer` constructor takes the original string to be separated as a parameter**

- **Each call to the `nextToken` method returns the next token in the string**



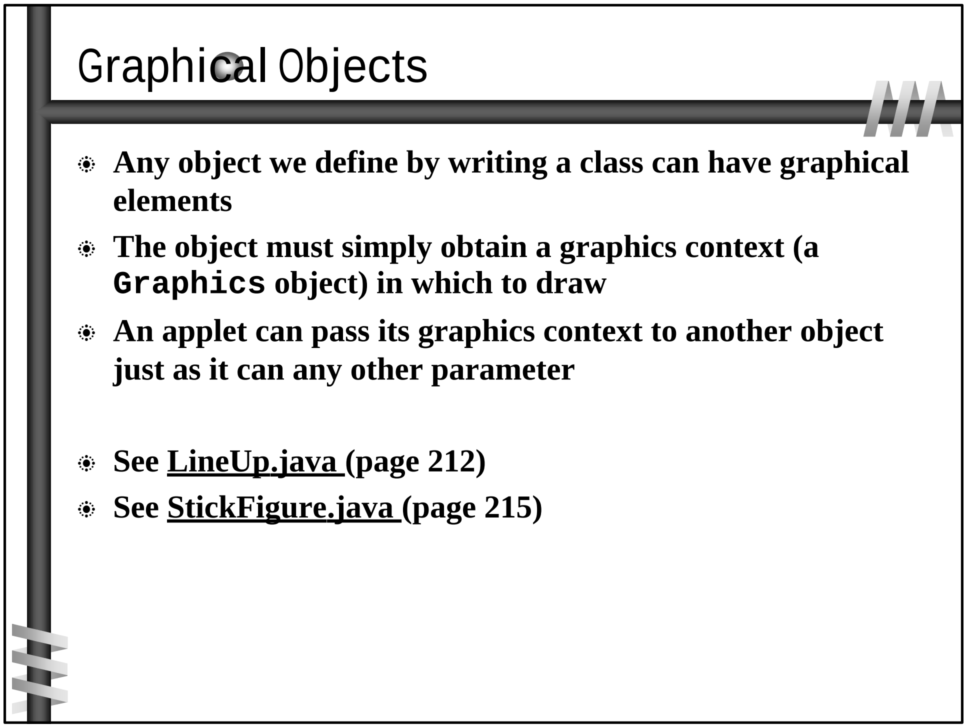
Method Decomposition

- A method should be relatively small, so that it can be readily understood as a single entity
- A potentially large method should be decomposed into several smaller methods as needed for clarity
- Therefore, a service method of an object may call one or more support methods to accomplish its goal
- See PigLatin.java (page 207)
- See PigLatinTranslator.java (page 208)



Applet Methods

- In previous examples we've used the `paint` method of the `Applet` class to draw on an applet
- The `Applet` class has several methods that are invoked automatically at certain points in an applet's life
- The `init` method, for instance, is executed only once when the applet is initially loaded
- The `Applet` class also contains other methods that generally assist in applet processing



Graphical Objects

- **Any object we define by writing a class can have graphical elements**
- **The object must simply obtain a graphics context (a `Graphics` object) in which to draw**
- **An applet can pass its graphics context to another object just as it can any other parameter**

- **See LineUp.java (page 212)**
- **See StickFigure.java (page 215)**

Presentation slides for

Java Software Solutions

Foundations of Program Design
Second Edition

by John Lewis and William Loftus

Part II
(chapters 6 – 12)

Prepared for Java Programming 90.301
by Marjan Trutschl
mtrutsch@cs.uml.edu

Chapter 5: Enhancing Classes

Presentation slides for

Java Software Solutions

Foundations of Program Design

Second Edition

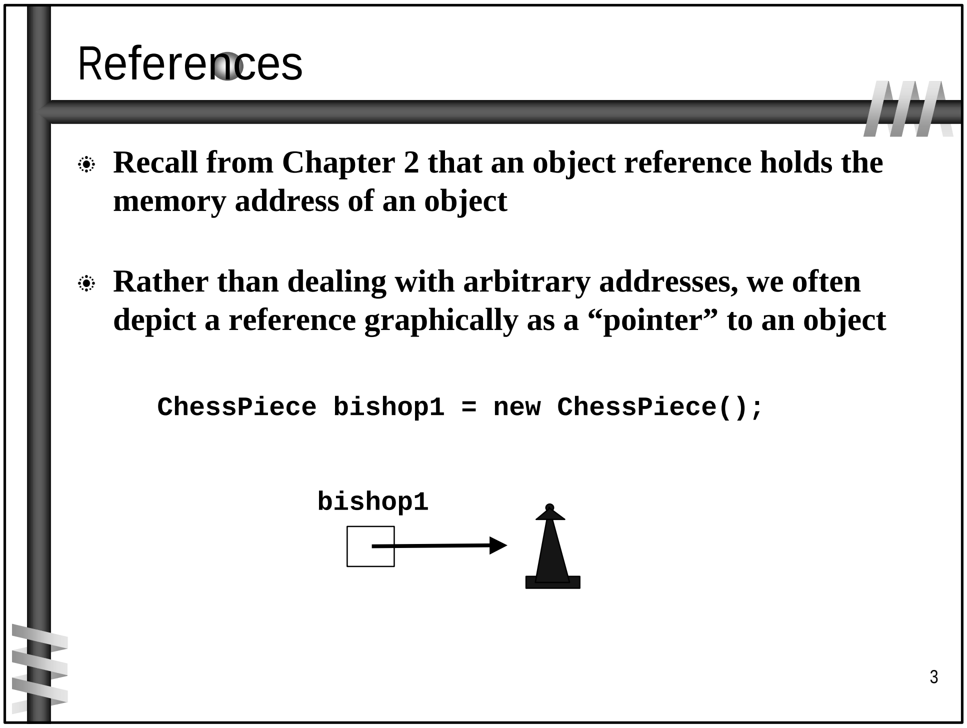
by John Lewis and William Loftus

Java Software Solutions is published by Addison-Wesley

Enhancing Classes

- **We can now explore various aspects of classes and objects in more detail**

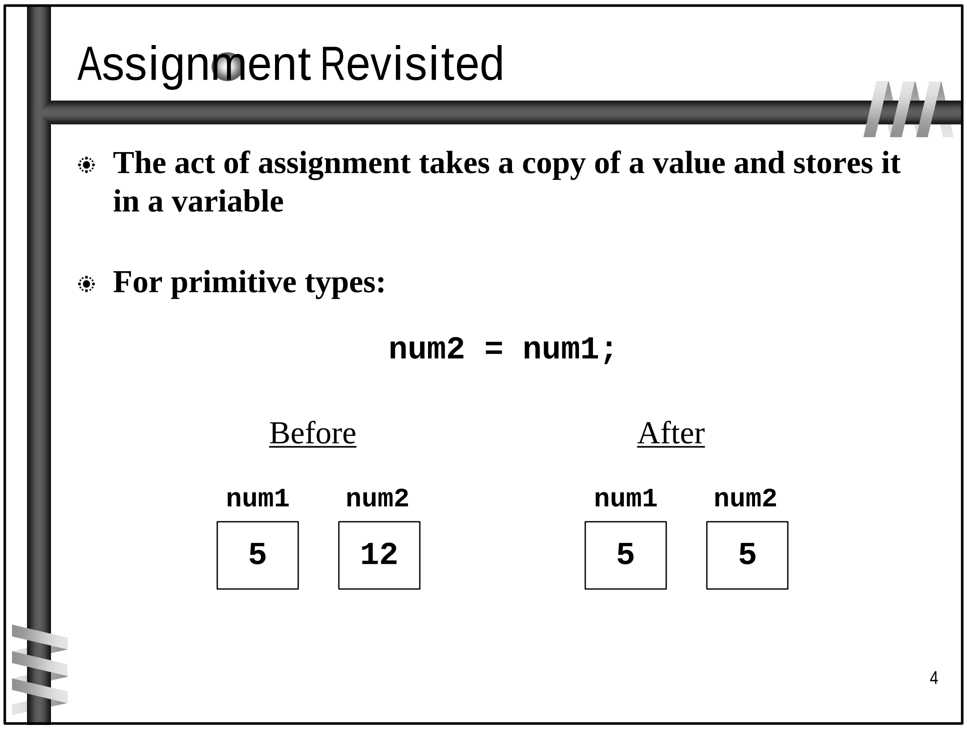
- **Chapter 5 focuses on:**
  - **object references and aliases**
  - **passing objects as parameters**
  - **the static modifier**
  - **nested classes**
  - **interfaces and polymorphism**
  - **events and listeners**
  - **animation**



References

- Recall from Chapter 2 that an object reference holds the memory address of an object
- Rather than dealing with arbitrary addresses, we often depict a reference graphically as a “pointer” to an object

```
ChessPiece bishop1 = new ChessPiece();
```

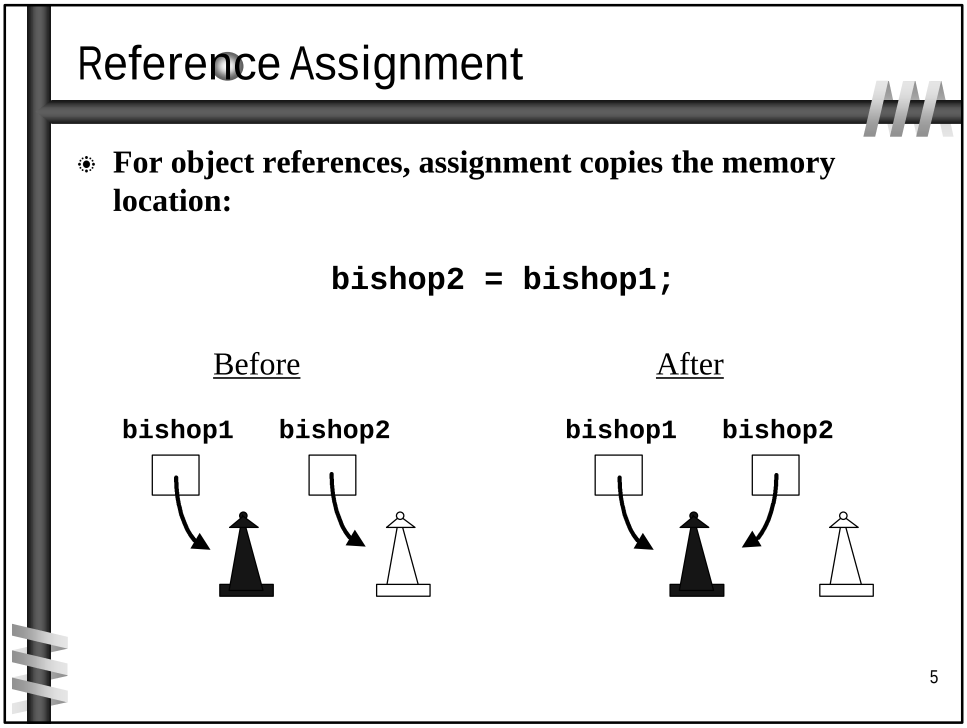


Assignment Revisited

- The act of assignment takes a copy of a value and stores it in a variable
- For primitive types:

```
num2 = num1;
```

Before
After
num1
num2
5
12
num1
num2
5
5



Reference Assignment

- **For object references, assignment copies the memory location:**

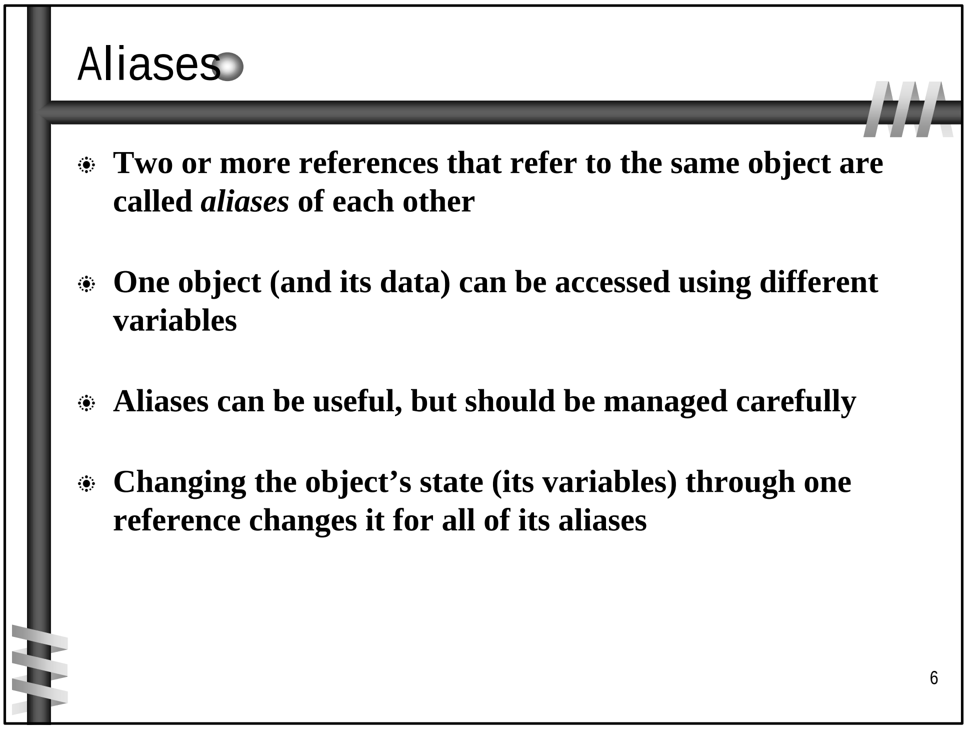
```
bishop2 = bishop1;
```

<u>Before</u>

bishop1 **bishop2**

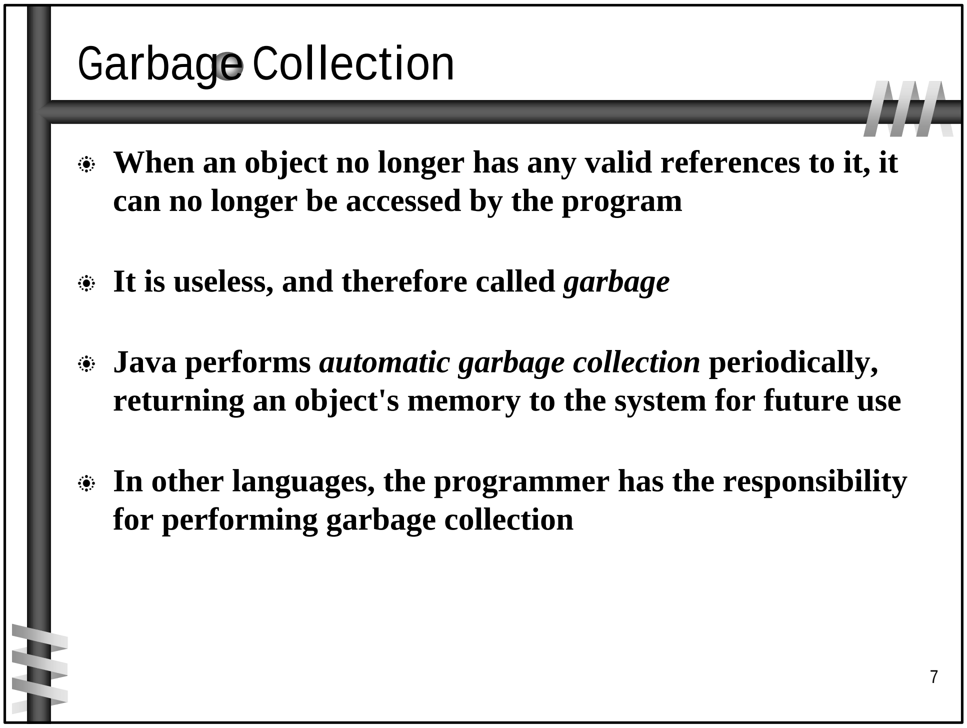
<u>After</u>

bishop1 **bishop2**



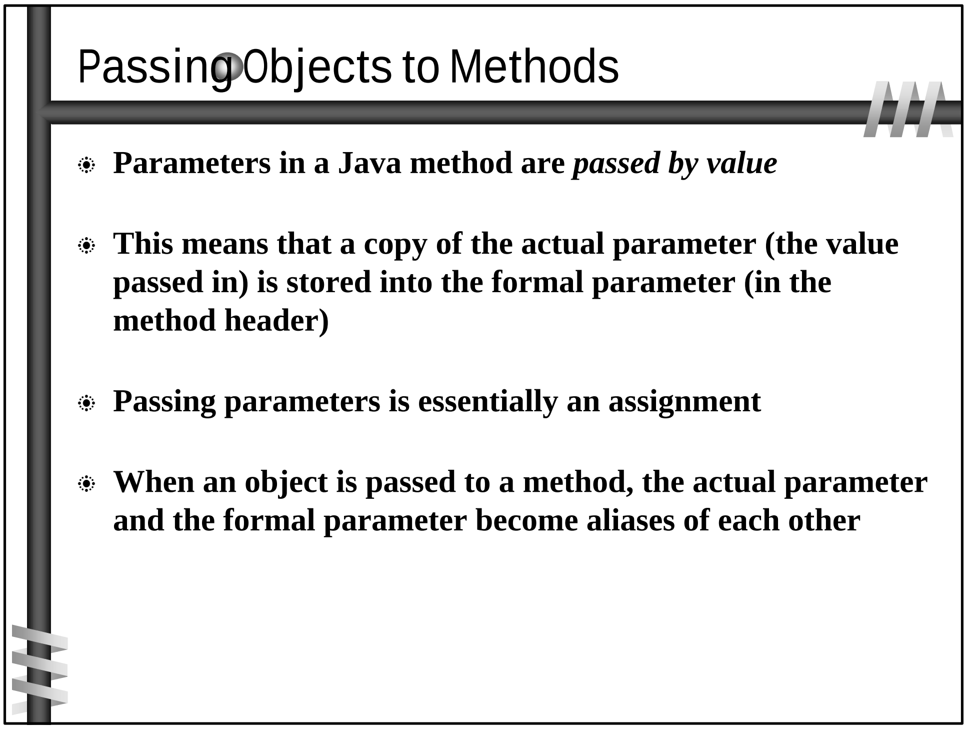
Aliases

- **Two or more references that refer to the same object are called *aliases* of each other**
- **One object (and its data) can be accessed using different variables**
- **Aliases can be useful, but should be managed carefully**
- **Changing the object's state (its variables) through one reference changes it for all of its aliases**



Garbage Collection

- When an object no longer has any valid references to it, it can no longer be accessed by the program
- It is useless, and therefore called *garbage*
- Java performs *automatic garbage collection* periodically, returning an object's memory to the system for future use
- In other languages, the programmer has the responsibility for performing garbage collection



Passing Objects to Methods

- Parameters in a Java method are *passed by value*
- This means that a copy of the actual parameter (the value passed in) is stored into the formal parameter (in the method header)
- Passing parameters is essentially an assignment
- When an object is passed to a method, the actual parameter and the formal parameter become aliases of each other

Passing Objects to Methods

- What you do to a parameter inside a method may or may not have a permanent effect (outside the method)

- See ParameterPassing.java (page 226)
- See ParameterTester.java (page 228)
- See Num.java (page 230)

- Note the difference between changing the reference and changing the object that the reference points to

The static Modifier

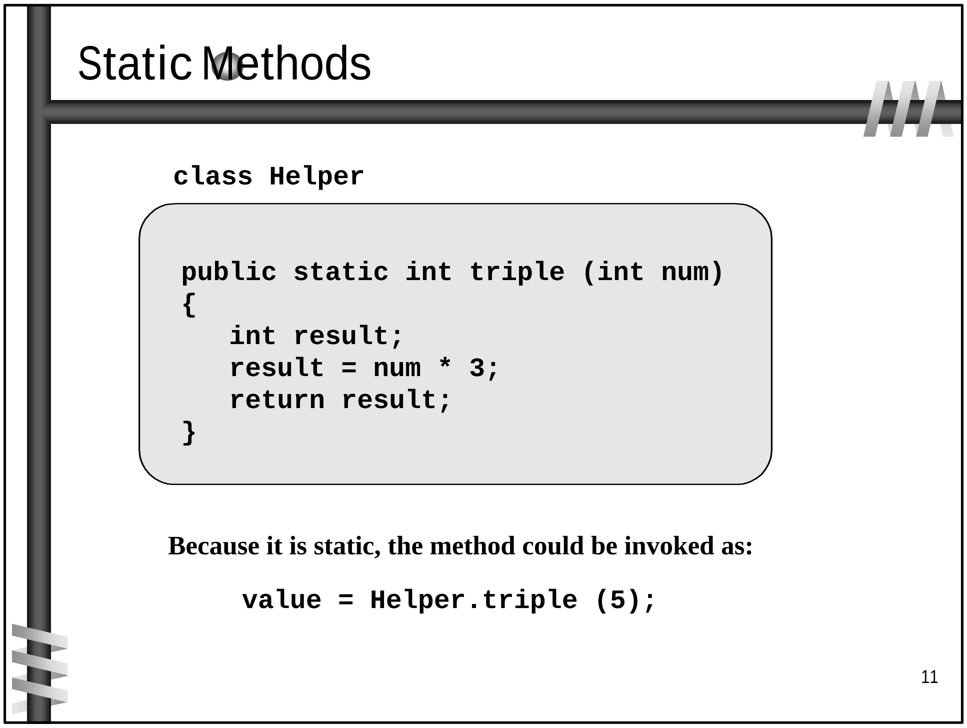
- In Chapter 2 we discussed static methods (also called class methods) that can be invoked through the class name rather than through a particular object

- For example, the methods of the `Math` class are static

- To make a method static, we apply the `static` modifier to the method definition

- The `static` modifier can be applied to variables as well

- It associates a variable or method with the class rather than an object



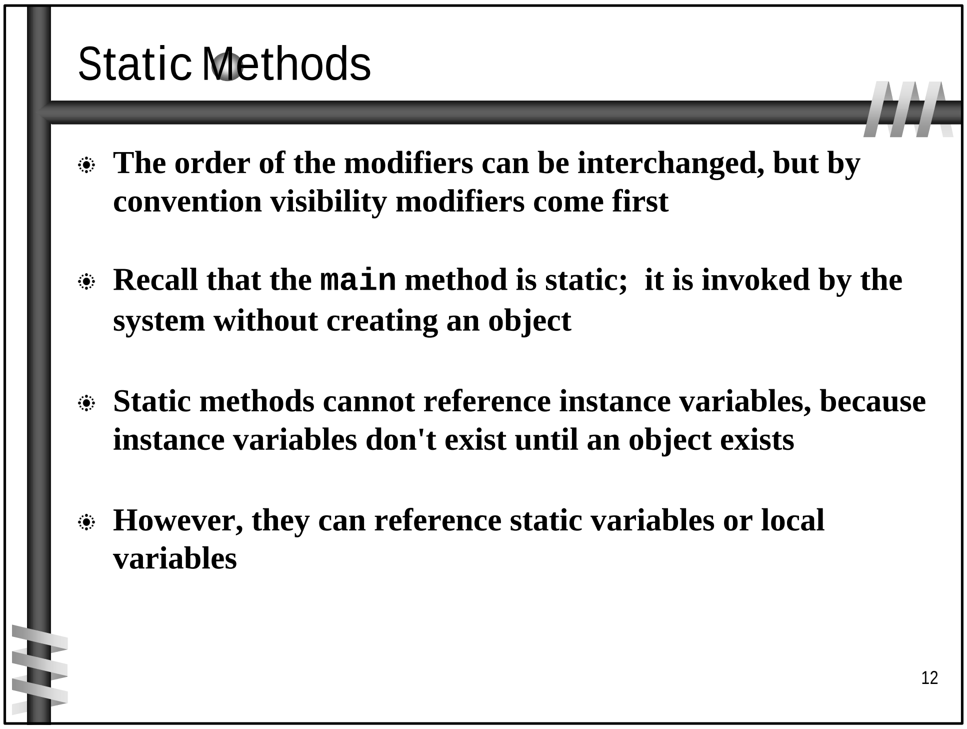
Static Methods

```
class Helper
```

```
public static int triple (int num)
{
   int result;
   result = num * 3;
   return result;
}
```

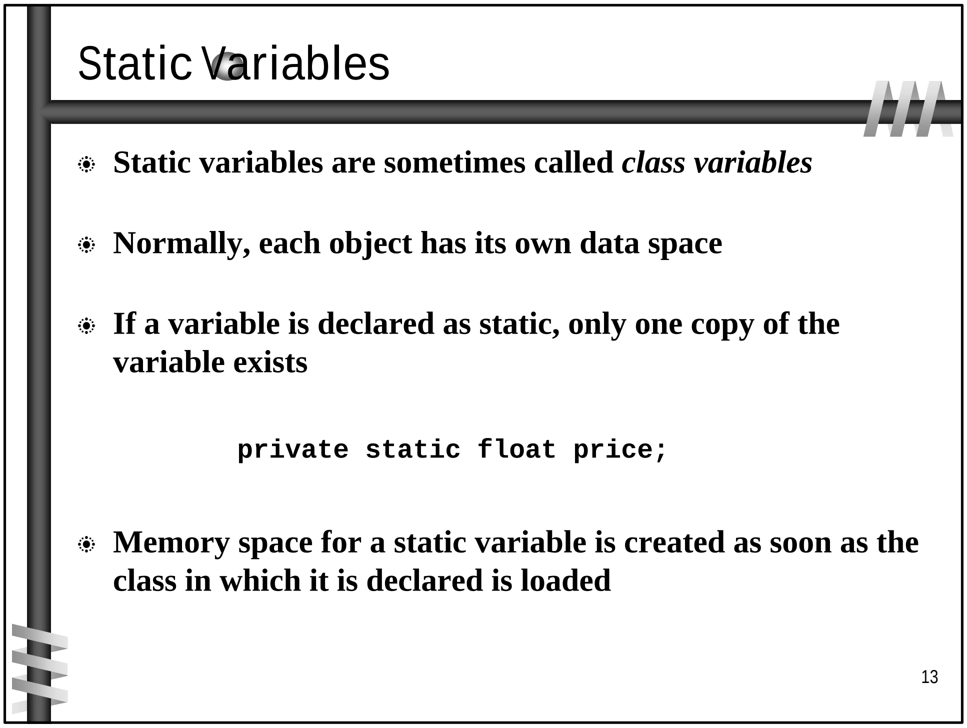
Because it is static, the method could be invoked as:

```
value = Helper.triple (5);
```



Static Methods

- **The order of the modifiers can be interchanged, but by convention visibility modifiers come first**
- **Recall that the `main` method is static; it is invoked by the system without creating an object**
- **Static methods cannot reference instance variables, because instance variables don't exist until an object exists**
- **However, they can reference static variables or local variables**



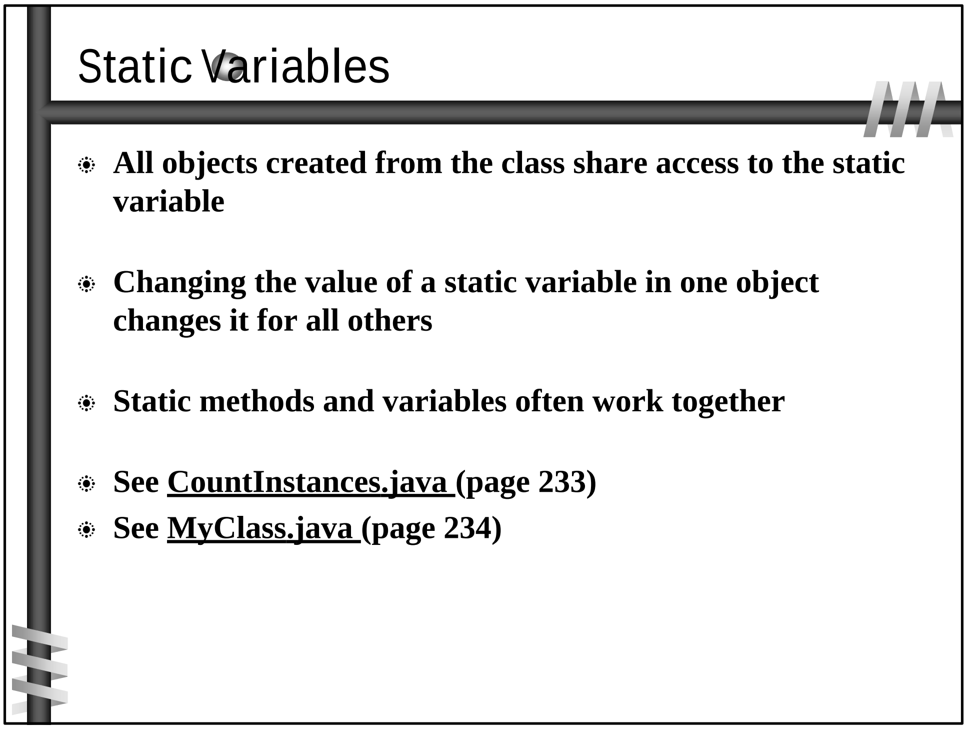
Static Variables

- **Static variables are sometimes called *class variables***
- **Normally, each object has its own data space**
- **If a variable is declared as static, only one copy of the variable exists**

```
        private static float price;
```

- **Memory space for a static variable is created as soon as the class in which it is declared is loaded**

13

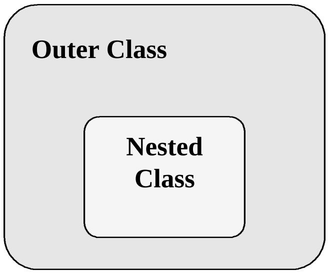


Static Variables

- **All objects created from the class share access to the static variable**
- **Changing the value of a static variable in one object changes it for all others**
- **Static methods and variables often work together**
- **See <u>CountInstances.java</u> (page 233)**
- **See <u>MyClass.java</u> (page 234)**

Nested Classes

- In addition to a class containing data and methods, it can also contain other classes
- A class declared within another class is called a *nested class*



Nested Classes

- A nested class has access to the variables and methods of the outer class, even if they are declared private
- In certain situations this makes the implementation of the classes easier because they can easily share information
- Furthermore, the nested class can be protected by the outer class from external use
- This is a special relationship and should be used with care

Nested Classes

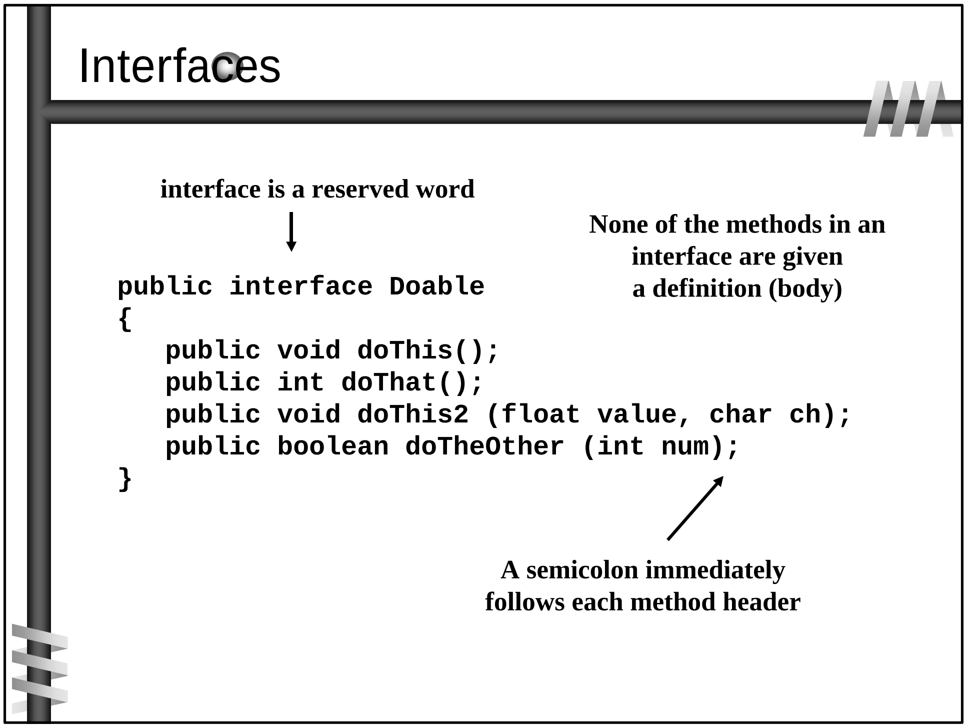
- A nested class produces a separate bytecode file
- If a nested class called Inside is declared in an outer class called Outside, two bytecode files will be produced:

```
Outside.class
Outside$Inside.class
```

- Nested classes can be declared as static, in which case they cannot refer to instance variables or methods
- A nonstatic nested class is called an *inner class*

Interfaces

- A Java *interface* is a collection of abstract methods and constants
- An *abstract method* is a method header without a method body
- An abstract method can be declared using the modifier `abstract`, but because all methods in an interface are abstract, it is usually left off
- An interface is used to formally define a set of methods that a class will implement



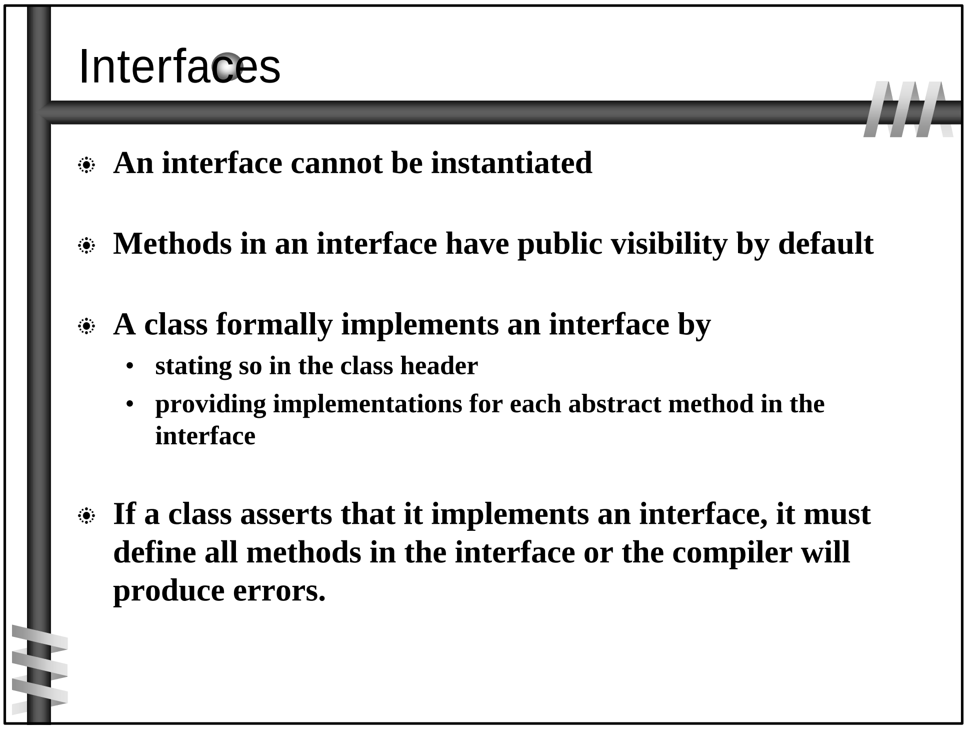
Interfaces

interface is a reserved word

None of the methods in an interface are given a definition (body)

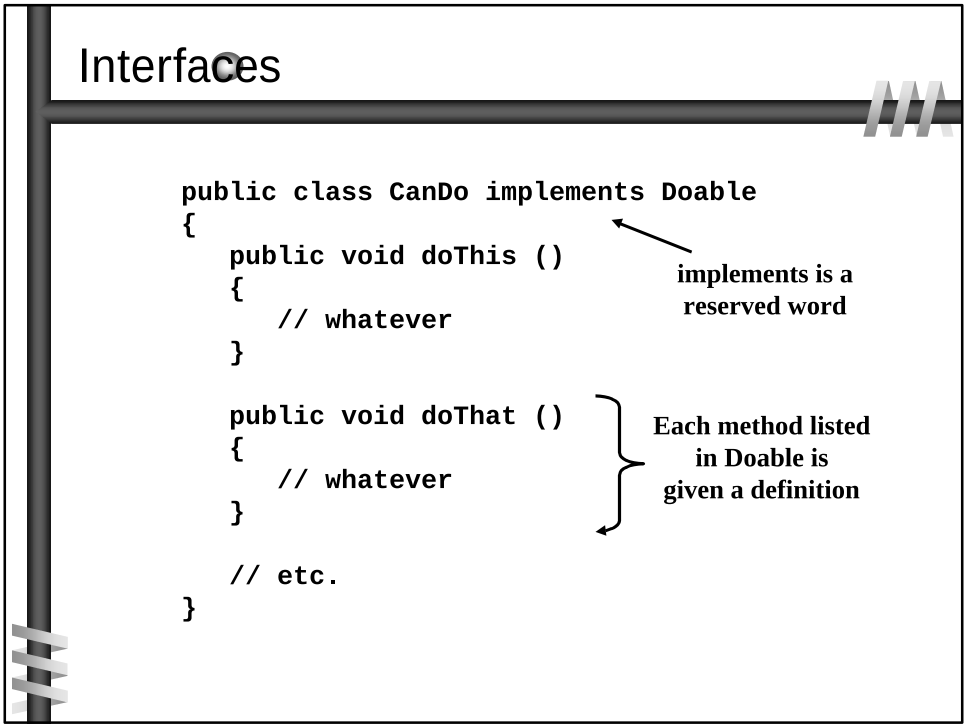
```
public interface Doable
{
   public void doThis();
   public int doThat();
   public void doThis2 (float value, char ch);
   public boolean doTheOther (int num);
}
```

A semicolon immediately follows each method header



Interfaces

- An interface cannot be instantiated
- Methods in an interface have public visibility by default
- A class formally implements an interface by
  - stating so in the class header
  - providing implementations for each abstract method in the interface
- If a class asserts that it implements an interface, it must define all methods in the interface or the compiler will produce errors.

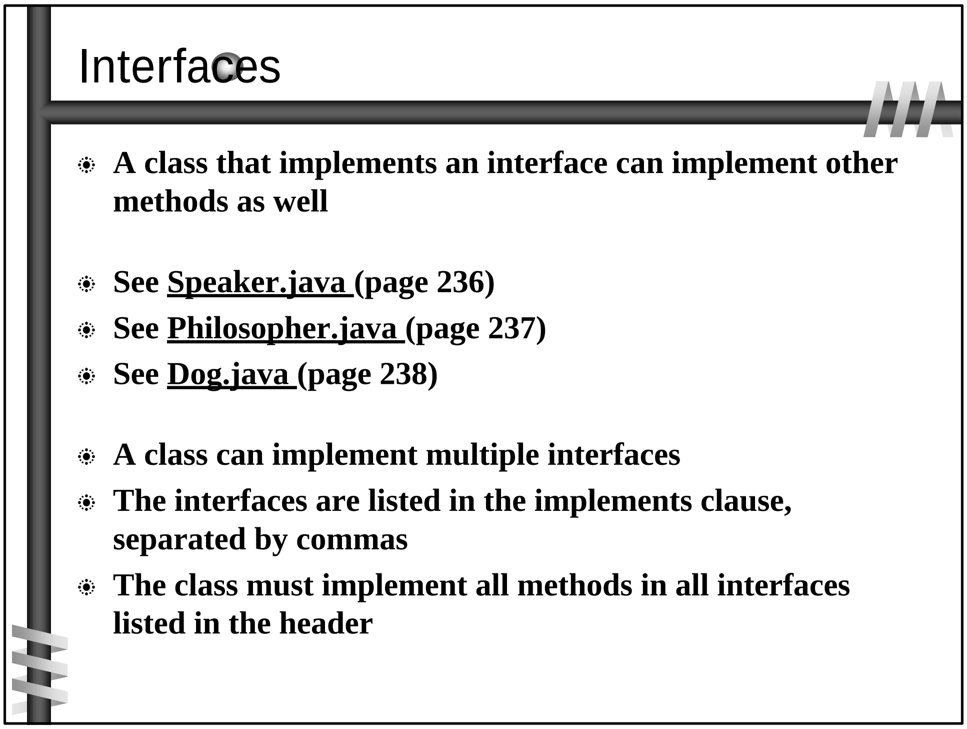


Interfaces

```
public class CanDo implements Doable
{
   public void doThis ()
   {
      // whatever
   }

   public void doThat ()
   {
      // whatever
   }

   // etc.
}
```

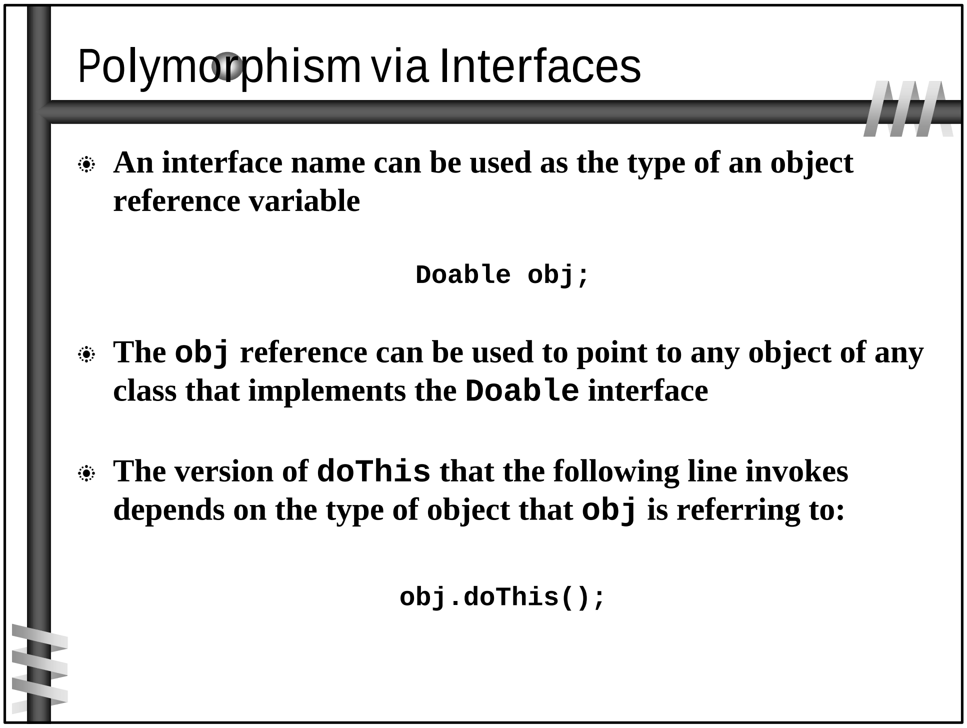
implements is a reserved word

Each method listed in Doable is given a definition



Interfaces

- **A class that implements an interface can implement other methods as well**
- **See <u>Speaker.java</u> (page 236)**
- **See <u>Philosopher.java</u> (page 237)**
- **See <u>Dog.java</u> (page 238)**
- **A class can implement multiple interfaces**
- **The interfaces are listed in the implements clause, separated by commas**
- **The class must implement all methods in all interfaces listed in the header**



Polymorphism via Interfaces

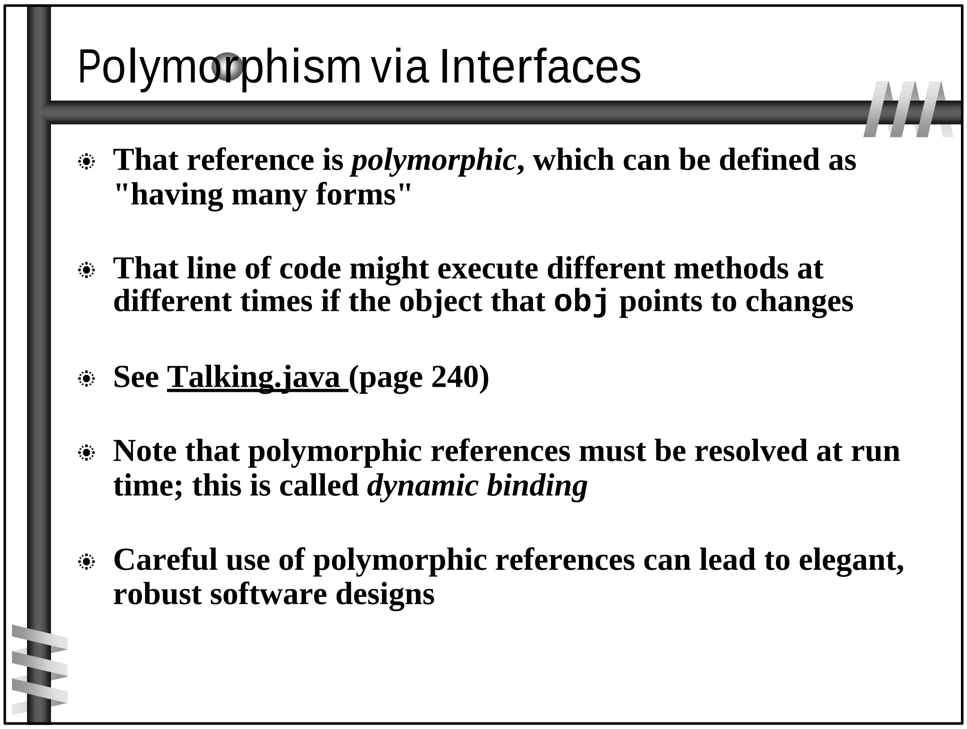
- An interface name can be used as the type of an object reference variable

```
Doable obj;
```

- The `obj` reference can be used to point to any object of any class that implements the `Doable` interface

- The version of `doThis` that the following line invokes depends on the type of object that `obj` is referring to:

```
obj.doThis();
```



Polymorphism via Interfaces

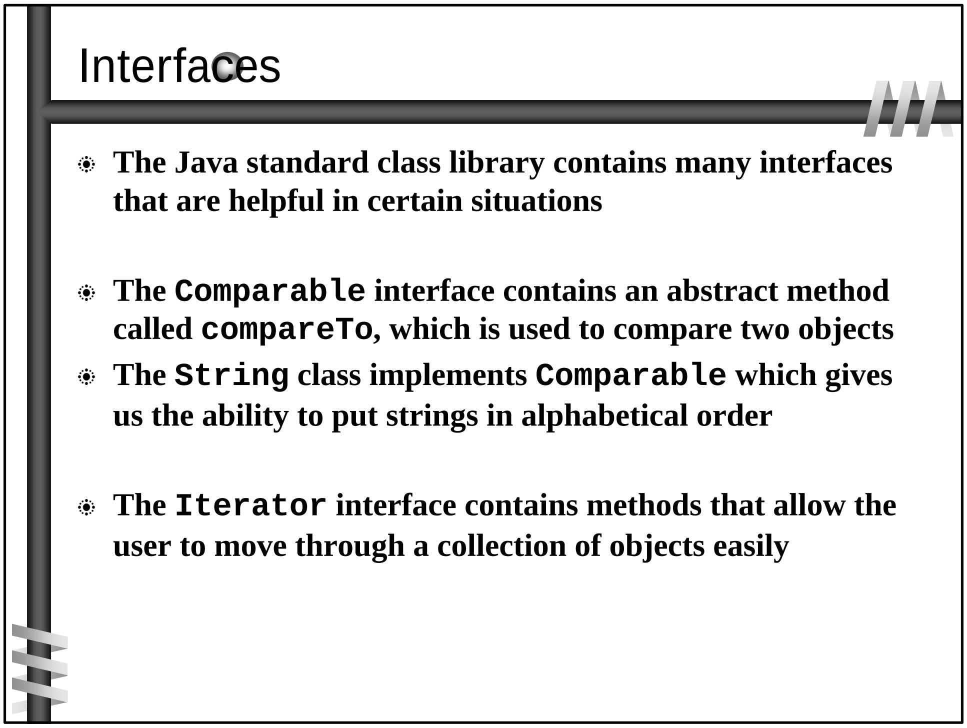
- That reference is *polymorphic*, which can be defined as "having many forms"

- That line of code might execute different methods at different times if the object that `obj` points to changes

- See Talking.java (page 240)

- Note that polymorphic references must be resolved at run time; this is called *dynamic binding*

- Careful use of polymorphic references can lead to elegant, robust software designs

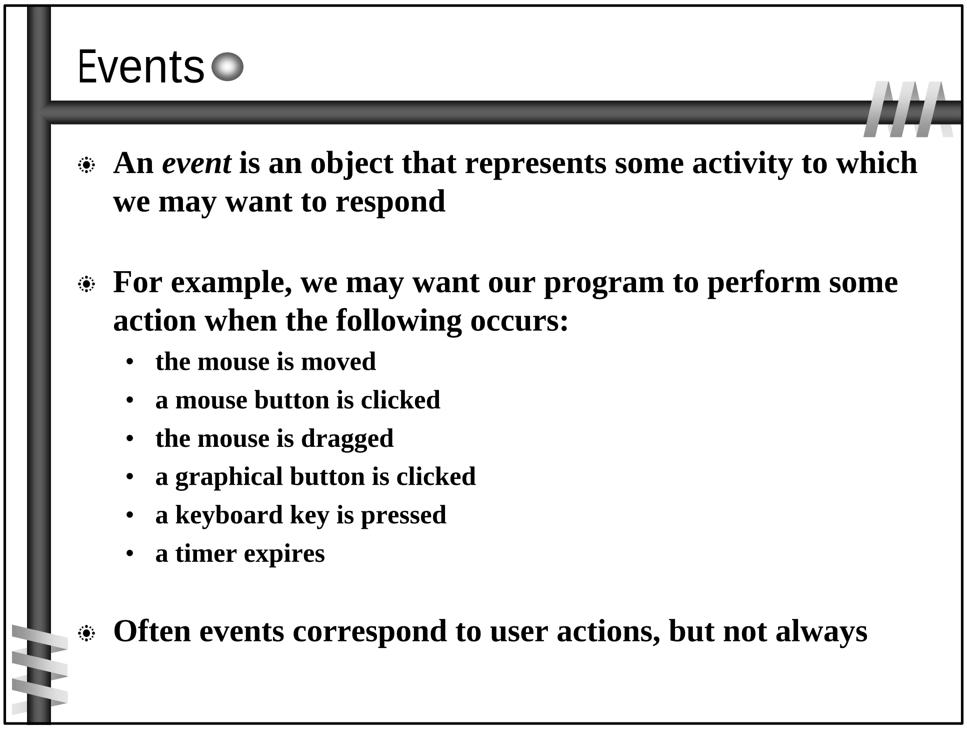


Interfaces

- The Java standard class library contains many interfaces that are helpful in certain situations

- The `Comparable` interface contains an abstract method called `compareTo`, which is used to compare two objects
- The `String` class implements `Comparable` which gives us the ability to put strings in alphabetical order

- The `Iterator` interface contains methods that allow the user to move through a collection of objects easily



Events

- An *event* is an object that represents some activity to which we may want to respond

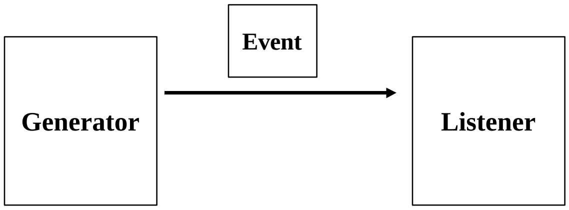
- For example, we may want our program to perform some action when the following occurs:
  - the mouse is moved
  - a mouse button is clicked
  - the mouse is dragged
  - a graphical button is clicked
  - a keyboard key is pressed
  - a timer expires

- Often events correspond to user actions, but not always

Events

- The Java standard class library contains several classes that represent typical events
- Certain objects, such as an applet or a graphical button, generate (fire) an event when it occurs
- Other objects, called *listeners*, respond to events
- We can write listener objects to do whatever we want when an event occurs

Events and Listeners



This object may generate an event

This object waits for and responds to an event

When an event occurs, the generator calls the appropriate method of the listener, passing an object that describes the event

Listener Interfaces

- We can create a listener object by writing a class that implements a particular *listener interface*
- The Java standard class library contains several interfaces that correspond to particular event categories
- For example, the `MouseListener` interface contains methods that correspond to mouse events
- After creating the listener, we *add* the listener to the component that might generate the event to set up a formal relationship between the generator and listener

Mouse Events

- The following are *mouse events*:
  - *mouse pressed* - the mouse button is pressed down
  - *mouse released* - the mouse button is released
  - *mouse clicked* - the mouse button is pressed and released
  - *mouse entered* - the mouse pointer is moved over a particular component
  - *mouse exited* - the mouse pointer is moved off of a particular component
- Any given program can listen for some, none, or all of these
- See Dots.java (page 246)
- See DotsMouseListener.java (page 248)

Mouse Motion Events

- The following are called *mouse motion events*:
  - *mouse moved* - the mouse is moved
  - *mouse dragged* - the mouse is moved while the mouse button is held down

- There is a corresponding `MouseMotionListener` interface
- One class can serve as both a generator and a listener
- One class can serve as a listener for multiple event types

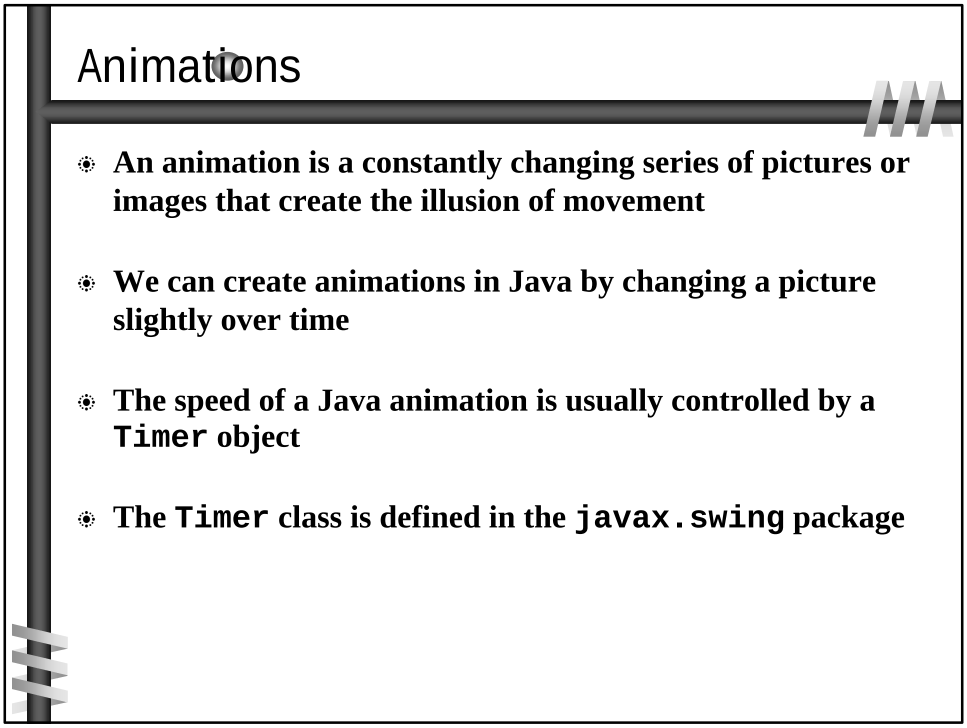
- See RubberLines.java (page 249)

Key Events

- The following are called *key events*:
  - *key pressed* - a keyboard key is pressed down
  - *key released* - a keyboard key is released
  - *key typed* - a keyboard key is pressed and released

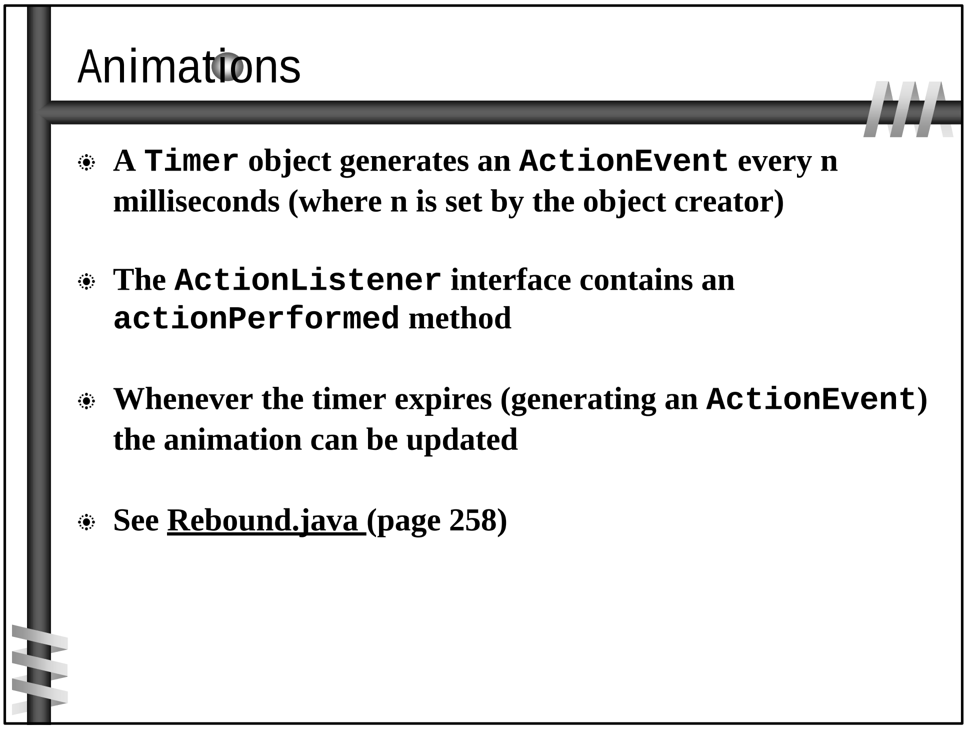
- The `KeyListener` interface handles key events
- Listener classes are often implemented as inner classes, nested within the component that they are listening to

- See Direction.java (page 253)



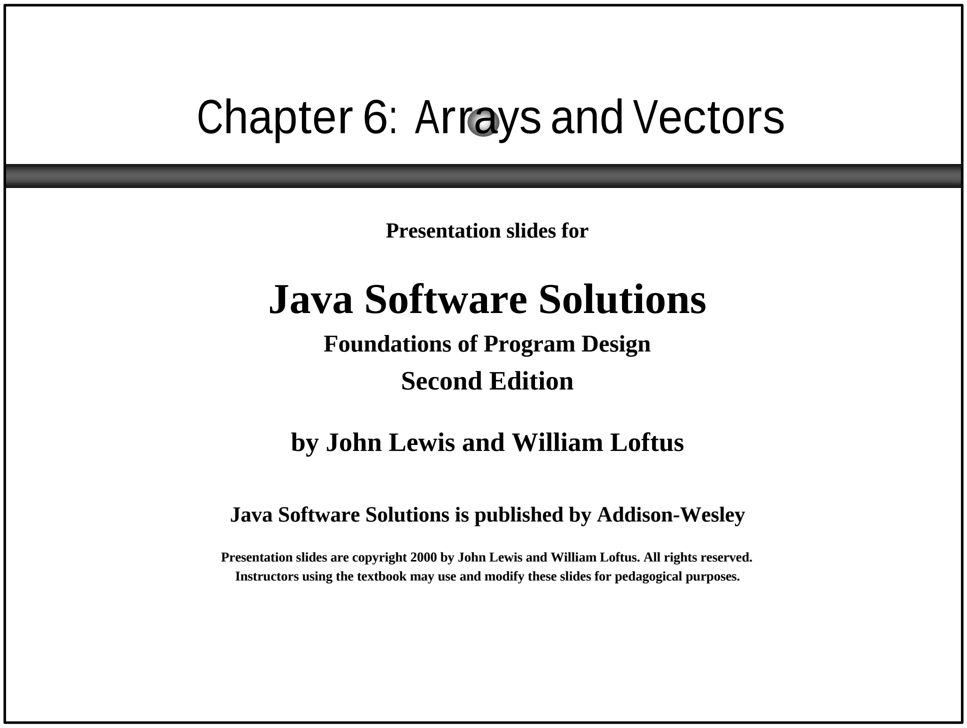
Animations

- An animation is a constantly changing series of pictures or images that create the illusion of movement
- We can create animations in Java by changing a picture slightly over time
- The speed of a Java animation is usually controlled by a `Timer` object
- The `Timer` class is defined in the `javax.swing` package



Animations

- A `Timer` object generates an `ActionEvent` every n milliseconds (where n is set by the object creator)
- The `ActionListener` interface contains an `actionPerformed` method
- Whenever the timer expires (generating an `ActionEvent`) the animation can be updated
- See <u>Rebound.java</u> (page 258)



Chapter 6: Arrays and Vectors

Presentation slides for

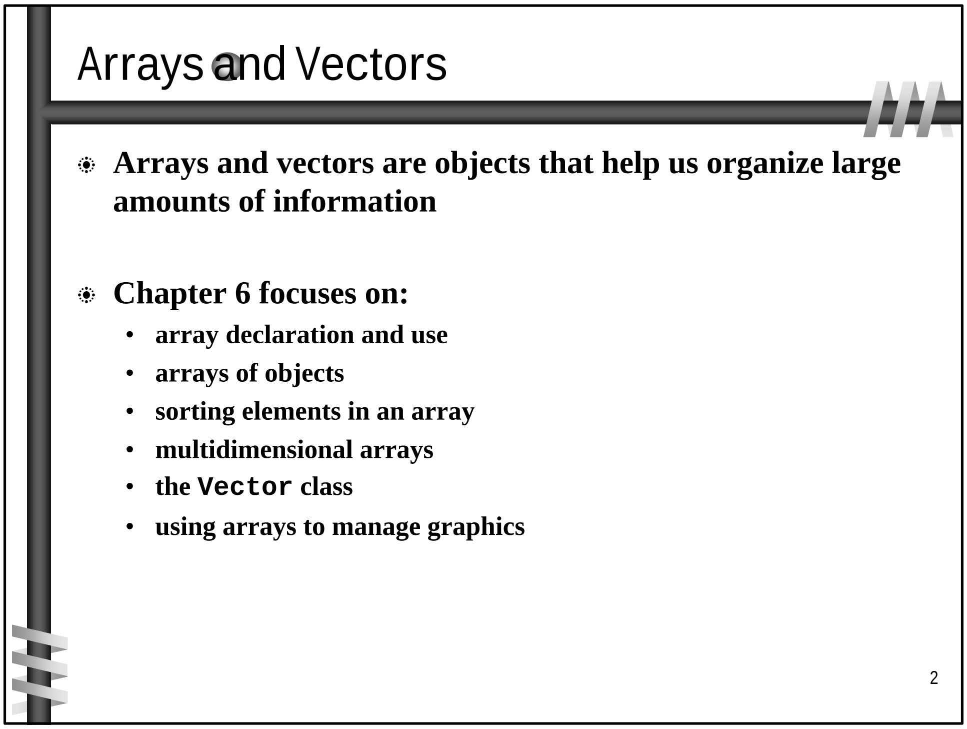
Java Software Solutions

Foundations of Program Design

Second Edition

by John Lewis and William Loftus

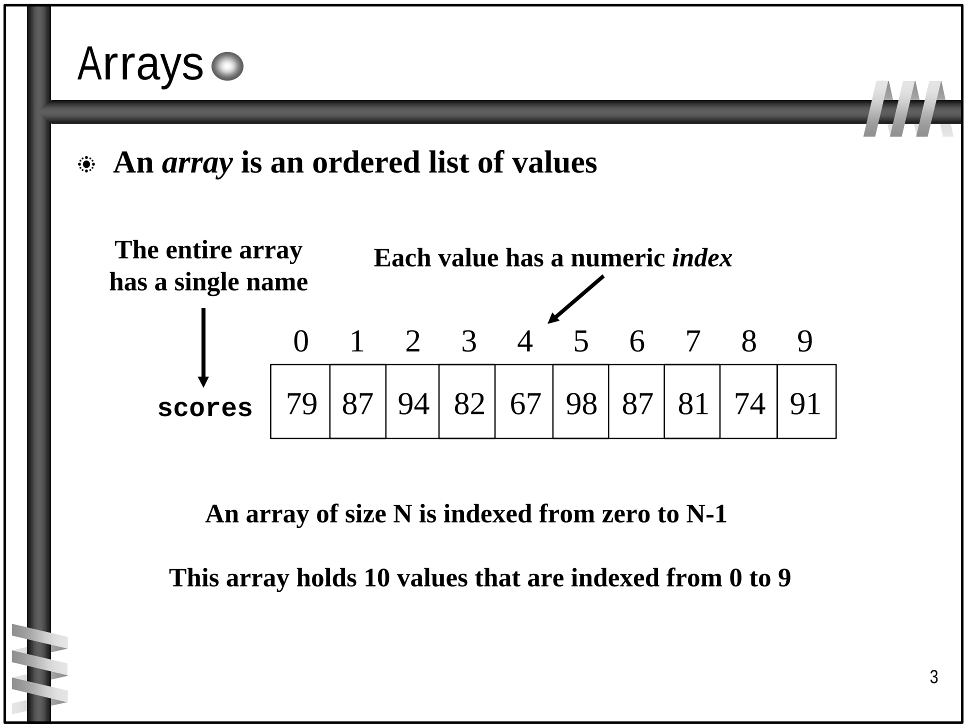
Java Software Solutions is published by Addison-Wesley



Arrays and Vectors

- **Arrays and vectors are objects that help us organize large amounts of information**

- **Chapter 6 focuses on:**
  - **array declaration and use**
  - **arrays of objects**
  - **sorting elements in an array**
  - **multidimensional arrays**
  - **the `Vector` class**
  - **using arrays to manage graphics**



Arrays

- An *array* is an ordered list of values

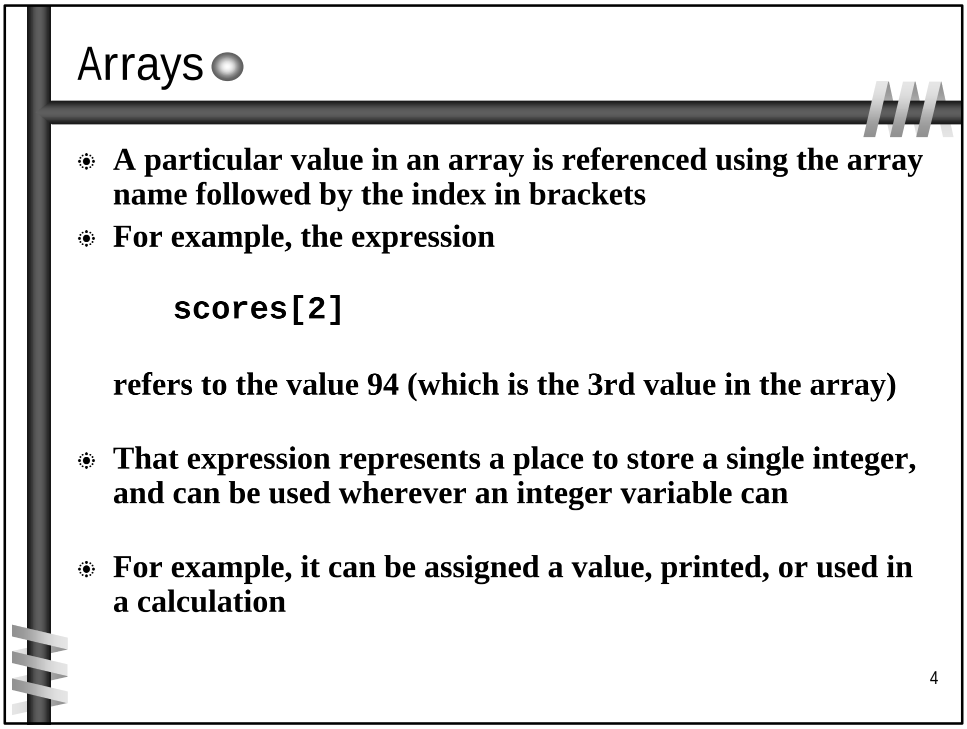
The entire array has a single name

Each value has a numeric *index*

	0	1	2	3	4	5	6	7	8	9
`scores`	79	87	94	82	67	98	87	81	74	91

An array of size N is indexed from zero to N-1

This array holds 10 values that are indexed from 0 to 9



Arrays

- A particular value in an array is referenced using the array name followed by the index in brackets
- For example, the expression

```
scores[2]
```

refers to the value 94 (which is the 3rd value in the array)

- That expression represents a place to store a single integer, and can be used wherever an integer variable can

- For example, it can be assigned a value, printed, or used in a calculation

Arrays

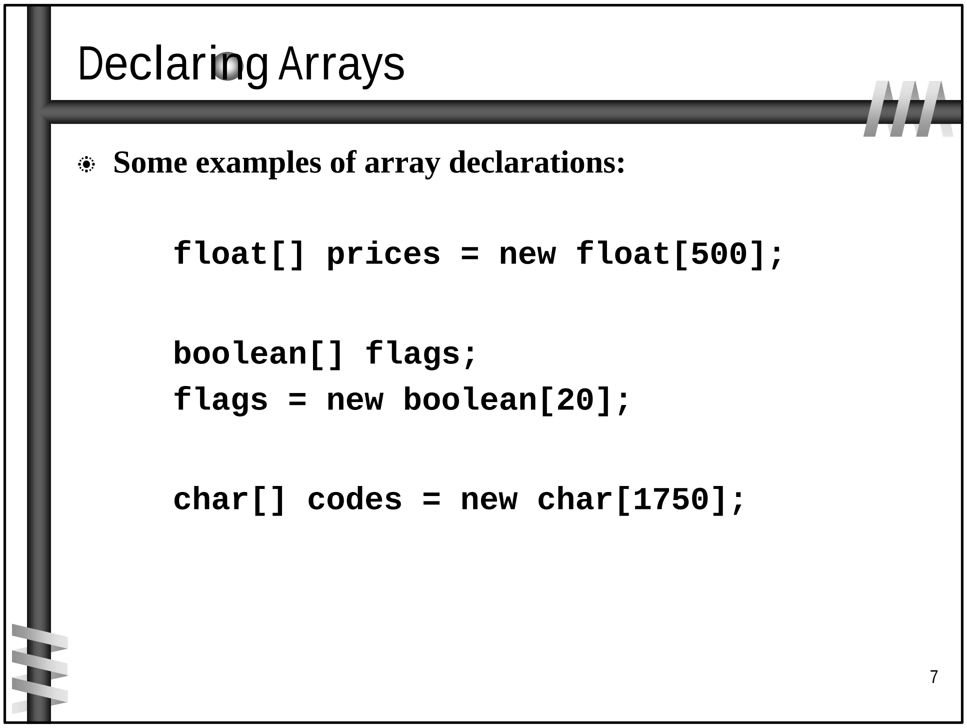
- An array stores multiple values of the same type
- That type can be primitive types or objects
- Therefore, we can create an array of integers, or an array of characters, or an array of String objects, etc.
- In Java, the array itself is an object
- Therefore the name of the array is a object reference variable, and the array itself is instantiated separately

Declaring Arrays

- The `scores` array could be declared as follows:

```
int[] scores = new int[10];
```

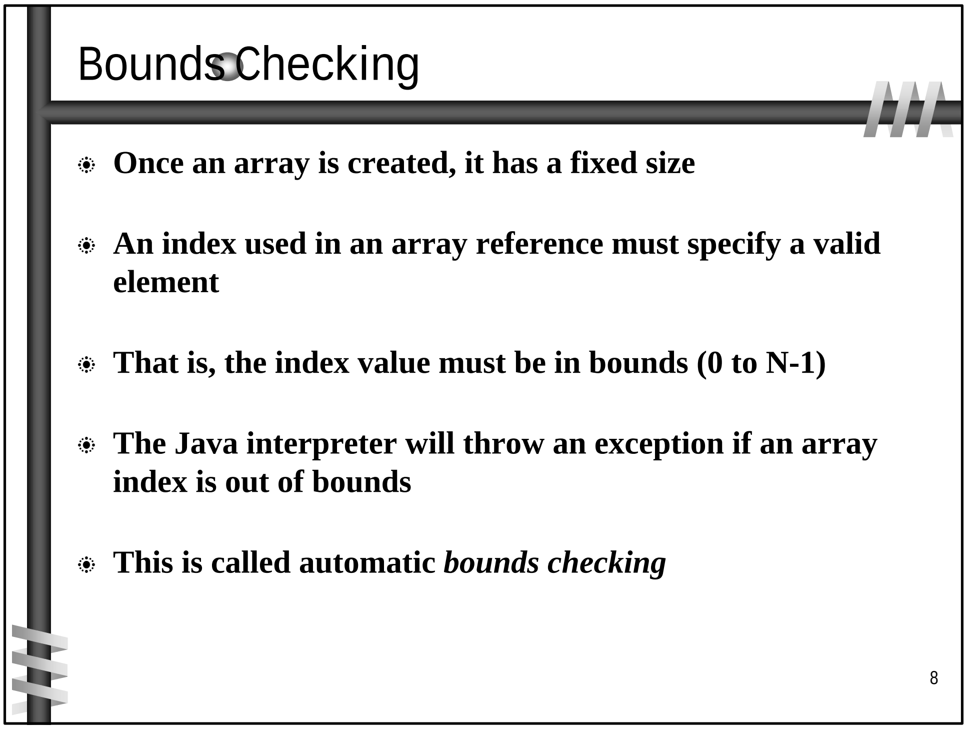
- Note that the type of the array does not specify its size, but each object of that type has a specific size
- The type of the variable `scores` is `int[]` (an array of integers)
- It is set to a new array object that can hold 10 integers
- See `BasicArray.java` (page 270)



Declaring Arrays

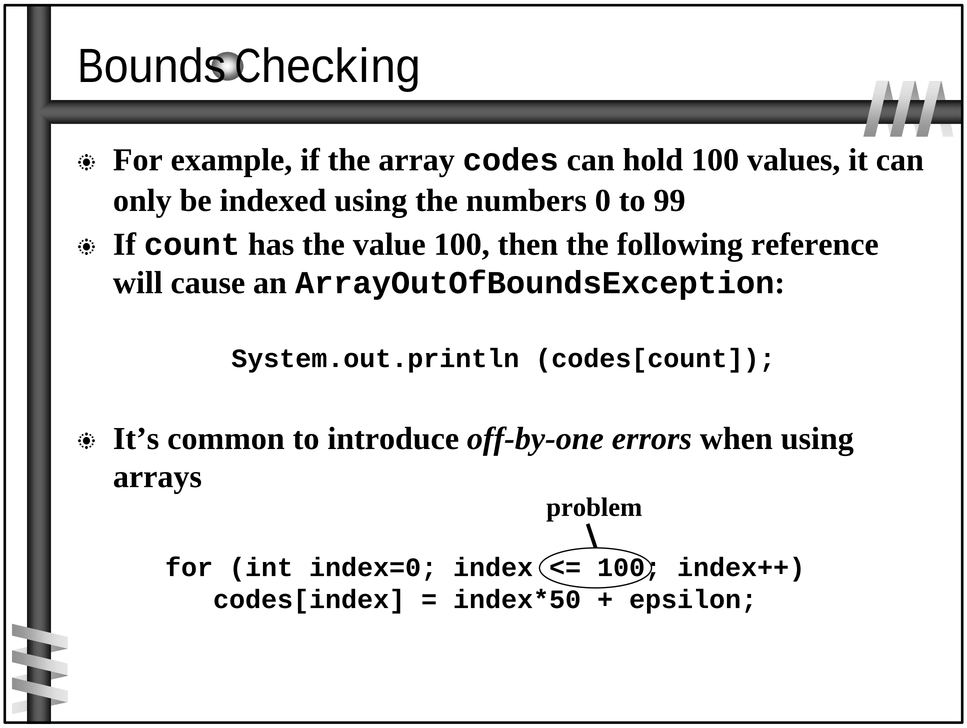
- Some examples of array declarations:

```
float[] prices = new float[500];

boolean[] flags;
flags = new boolean[20];

char[] codes = new char[1750];
```



Bounds Checking

- Once an array is created, it has a fixed size
- An index used in an array reference must specify a valid element
- That is, the index value must be in bounds (0 to N-1)
- The Java interpreter will throw an exception if an array index is out of bounds
- This is called automatic *bounds checking*



Bounds Checking

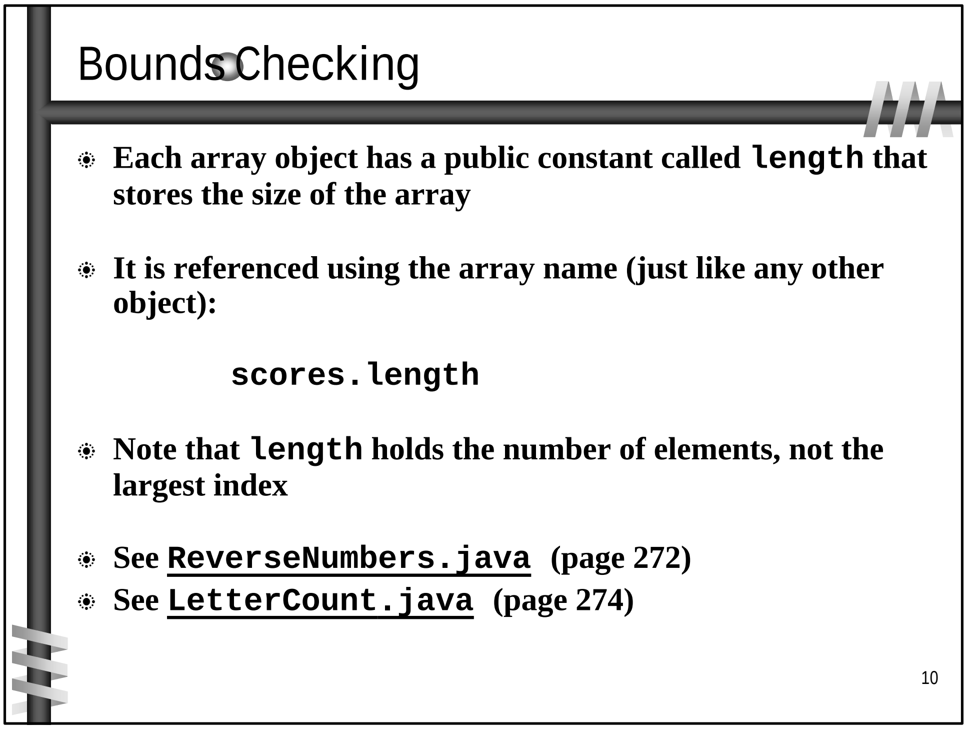
- For example, if the array `codes` can hold 100 values, it can only be indexed using the numbers 0 to 99
- If `count` has the value 100, then the following reference will cause an `ArrayOutOfBoundsException`:

```
System.out.println (codes[count]);
```

- It's common to introduce *off-by-one errors* when using arrays

problem

```
for (int index=0; index <= 100; index++)
   codes[index] = index*50 + epsilon;
```



Bounds Checking

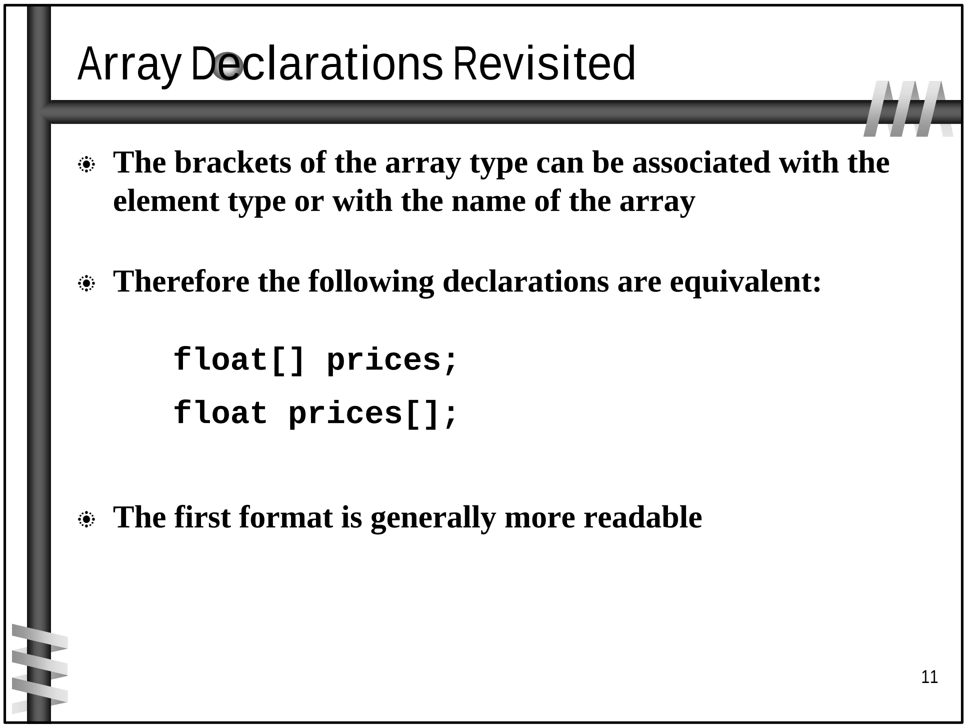
- Each array object has a public constant called `length` that stores the size of the array

- It is referenced using the array name (just like any other object):

```
scores.length
```

- Note that `length` holds the number of elements, not the largest index

- See `ReverseNumbers.java` (page 272)
- See `LetterCount.java` (page 274)

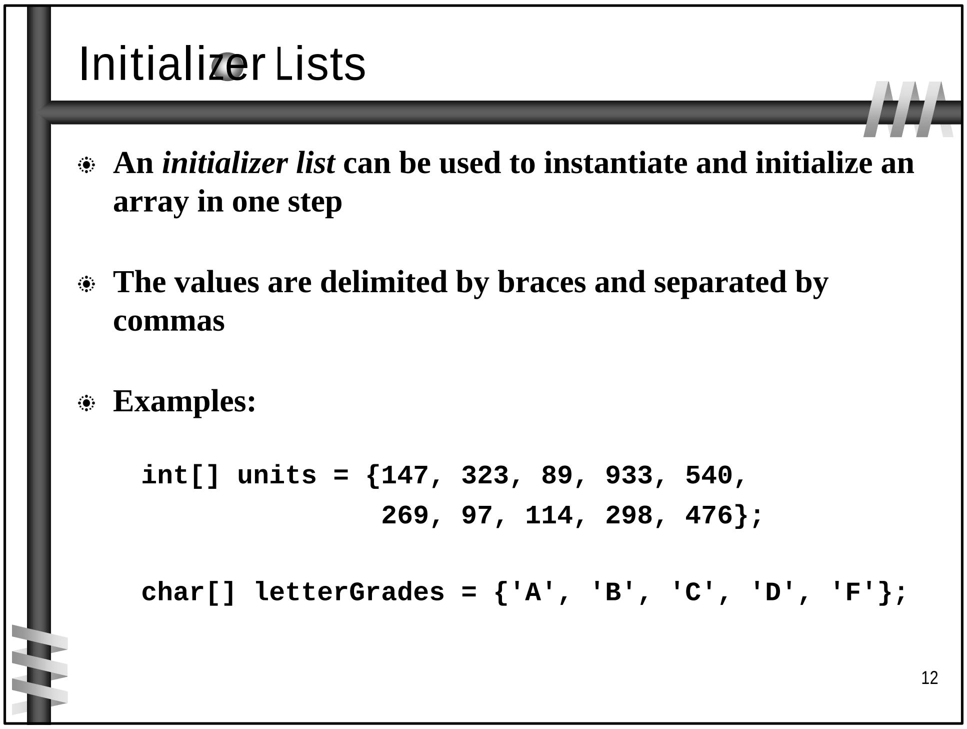


Array Declarations Revisited

- The brackets of the array type can be associated with the element type or with the name of the array
- Therefore the following declarations are equivalent:

```
float[] prices;
float prices[];
```

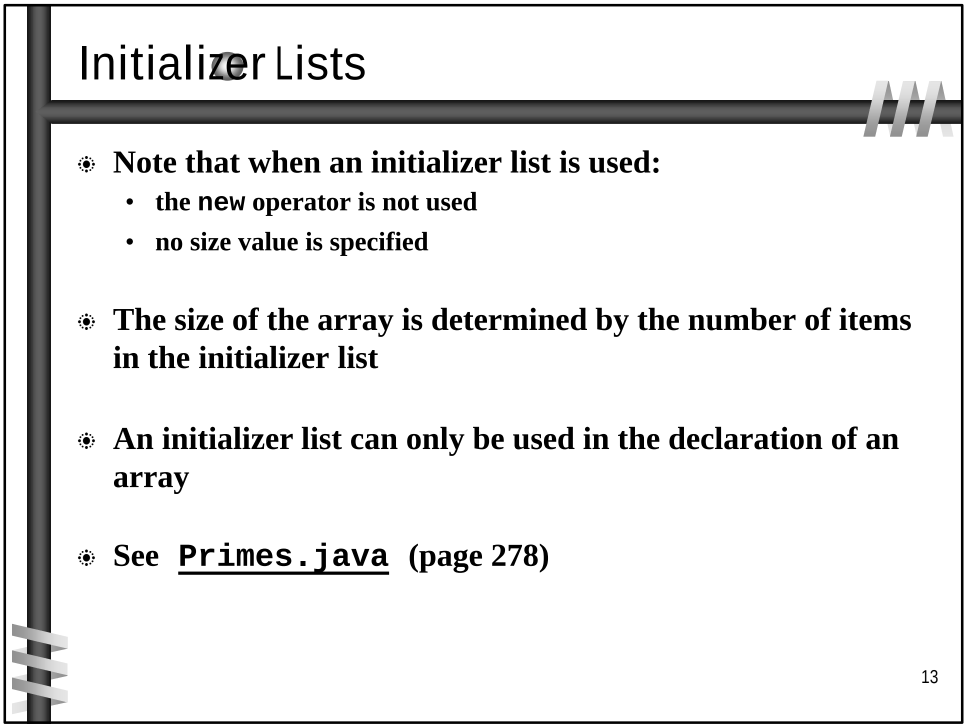
- The first format is generally more readable



Initializer Lists

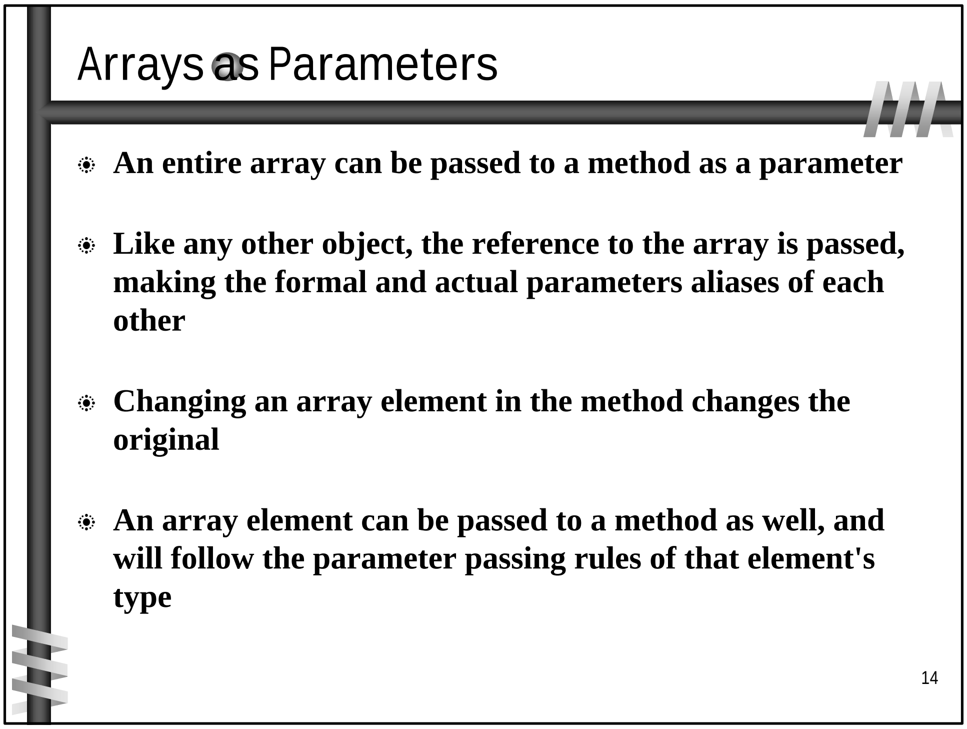
- An *initializer list* can be used to instantiate and initialize an array in one step
- The values are delimited by braces and separated by commas
- Examples:

```
int[] units = {147, 323, 89, 933, 540,
               269, 97, 114, 298, 476};

char[] letterGrades = {'A', 'B', 'C', 'D', 'F'};
```



Initializer Lists

- Note that when an initializer list is used:
  - the `new` operator is not used
  - no size value is specified
- The size of the array is determined by the number of items in the initializer list
- An initializer list can only be used in the declaration of an array
- See `Primes.java` (page 278)



Arrays as Parameters

- An entire array can be passed to a method as a parameter
- Like any other object, the reference to the array is passed, making the formal and actual parameters aliases of each other
- Changing an array element in the method changes the original
- An array element can be passed to a method as well, and will follow the parameter passing rules of that element's type

Arrays of Objects

- The elements of an array can be object references
- The following declaration reserves space to store 25 references to `String` objects

```
String[] words = new String[25];
```

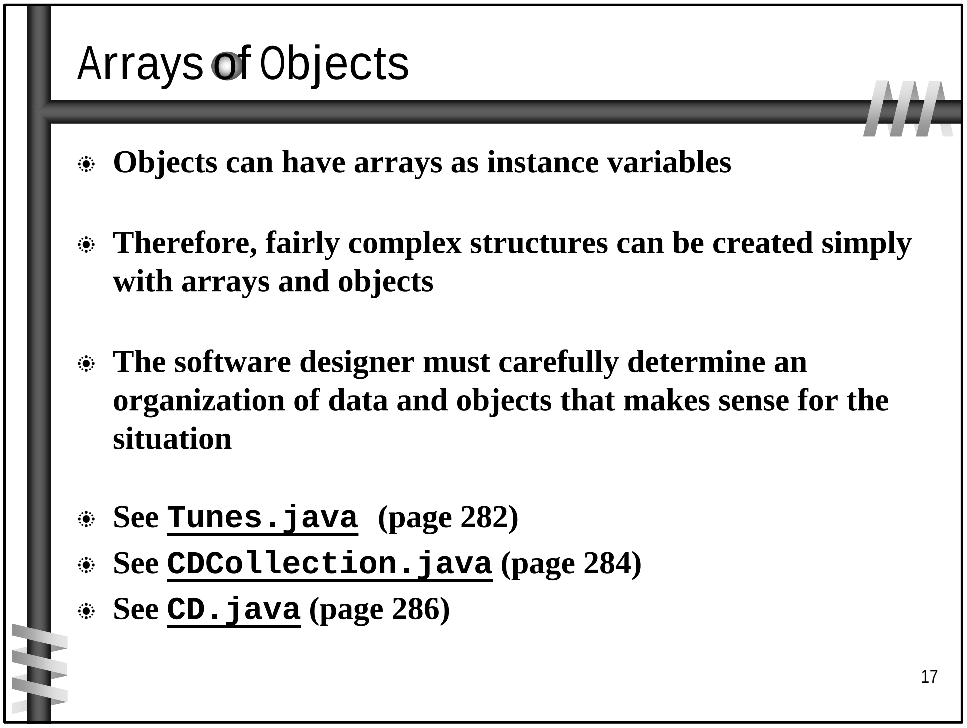
- It does NOT create the `String` objects themselves
- Each object stored in an array must be instantiated separately
- See `GradeRange.java` (page 280)

Command-Line Arguments

- The signature of the `main` method indicates that it takes an array of `String` objects as a parameter
- These values come from command-line arguments that are provided when the interpreter is invoked
- For example, the following invocation of the interpreter passes an array of three `String` objects into main:

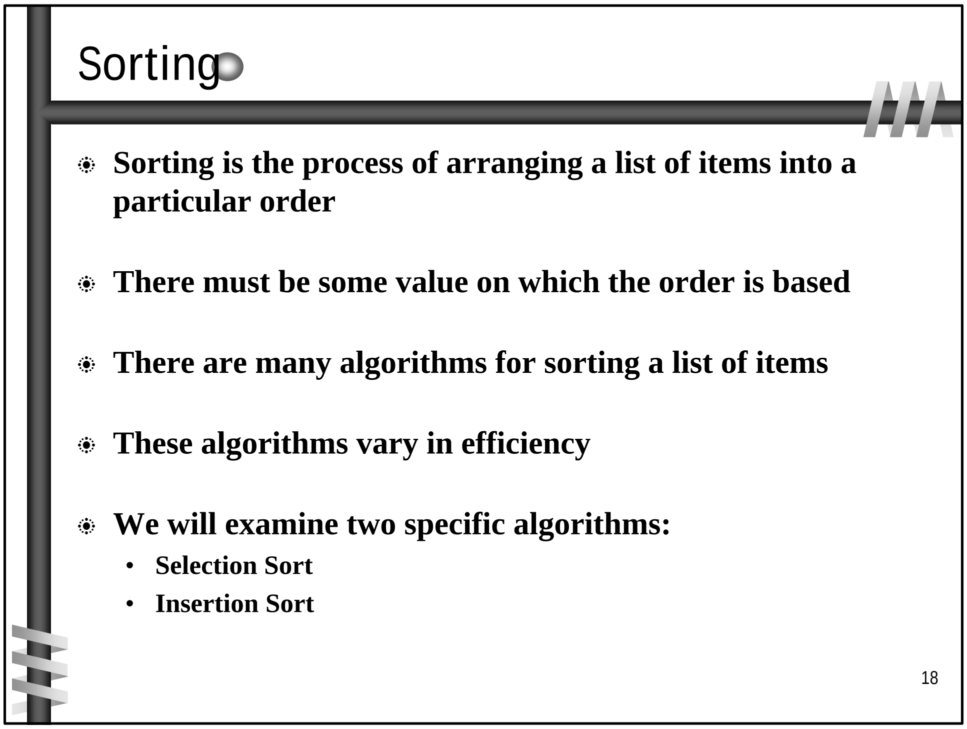
```
> java DoIt pennsylvania texas california
```

- These strings are stored at indexes 0-2 of the parameter
- See `NameTag.java` (page 281)



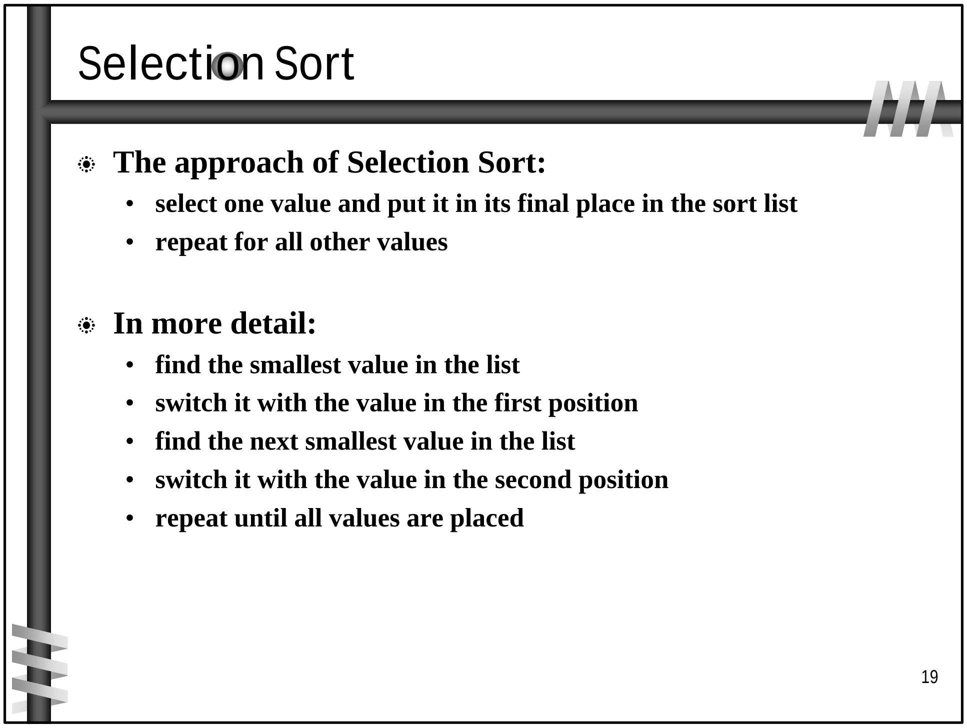
Arrays of Objects

- Objects can have arrays as instance variables
- Therefore, fairly complex structures can be created simply with arrays and objects
- The software designer must carefully determine an organization of data and objects that makes sense for the situation
- See `Tunes.java` (page 282)
- See `CDCollection.java` (page 284)
- See `CD.java` (page 286)



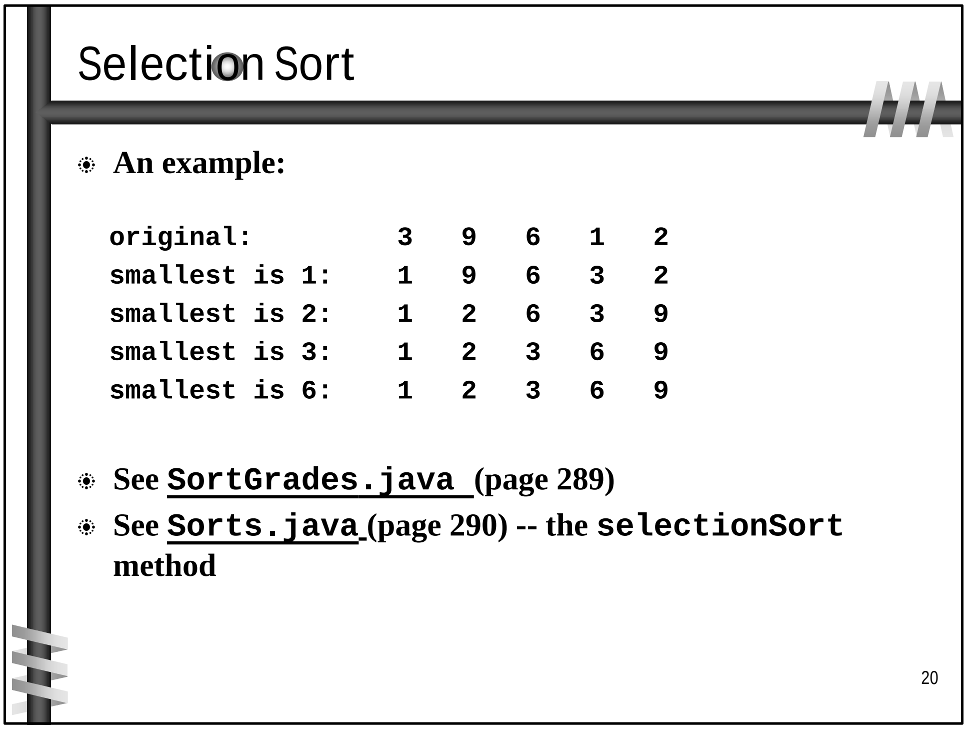
Sorting

- Sorting is the process of arranging a list of items into a particular order
- There must be some value on which the order is based
- There are many algorithms for sorting a list of items
- These algorithms vary in efficiency
- We will examine two specific algorithms:
  - Selection Sort
  - Insertion Sort



Selection Sort

- **The approach of Selection Sort:**
  - **select one value and put it in its final place in the sort list**
  - **repeat for all other values**
- **In more detail:**
  - **find the smallest value in the list**
  - **switch it with the value in the first position**
  - **find the next smallest value in the list**
  - **switch it with the value in the second position**
  - **repeat until all values are placed**

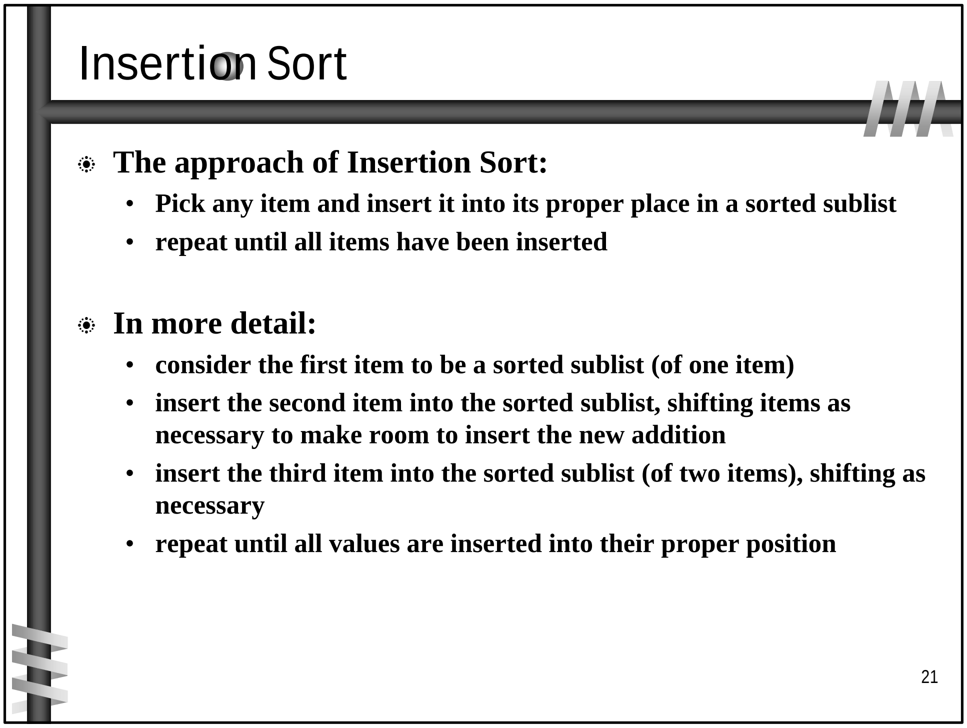


Selection Sort

- **An example:**

```
original:         3   9   6   1   2
smallest is 1:    1   9   6   3   2
smallest is 2:    1   2   6   3   9
smallest is 3:    1   2   3   6   9
smallest is 6:    1   2   3   6   9
```

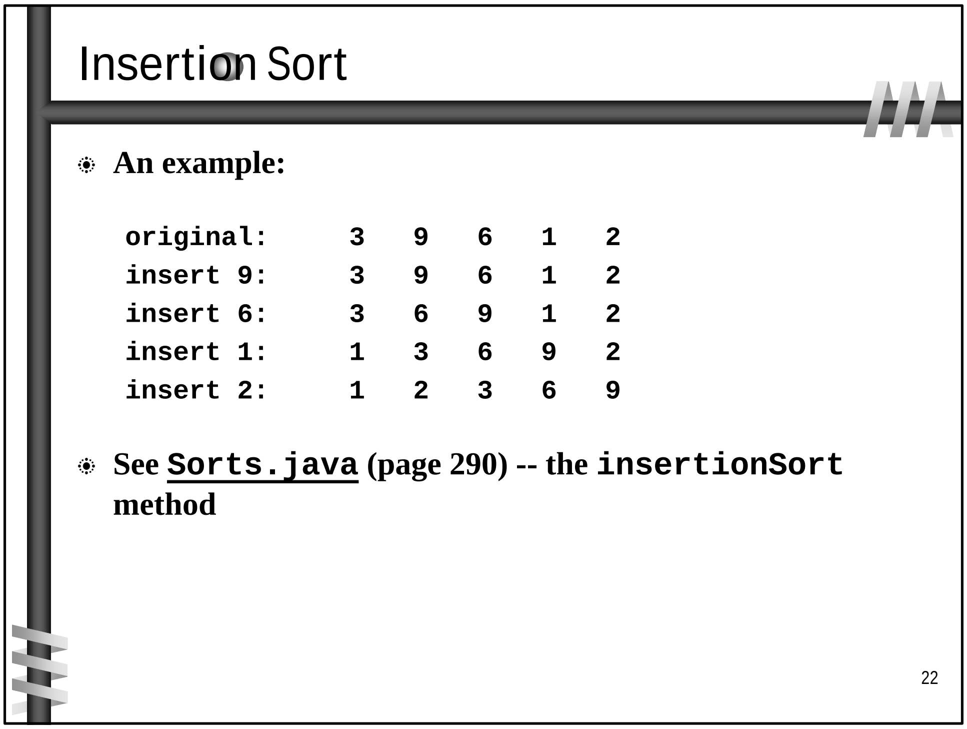
- **See `SortGrades.java` (page 289)**
- **See `Sorts.java` (page 290) -- the `selectionSort` method**



Insertion Sort

- **The approach of Insertion Sort:**
  - **Pick any item and insert it into its proper place in a sorted sublist**
  - **repeat until all items have been inserted**

- **In more detail:**
  - **consider the first item to be a sorted sublist (of one item)**
  - **insert the second item into the sorted sublist, shifting items as necessary to make room to insert the new addition**
  - **insert the third item into the sorted sublist (of two items), shifting as necessary**
  - **repeat until all values are inserted into their proper position**

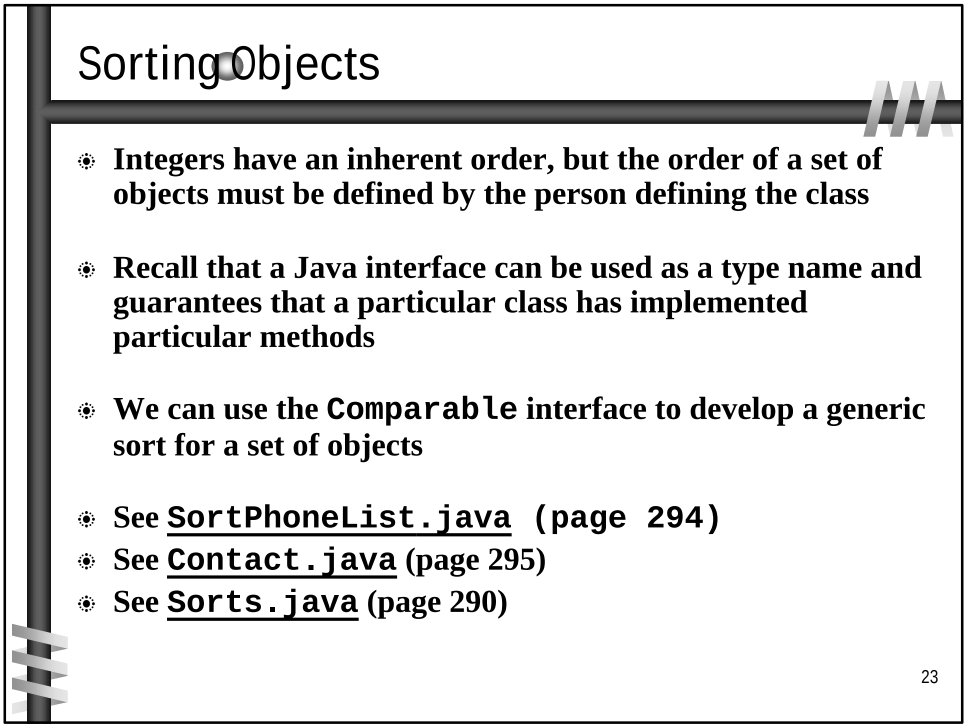


Insertion Sort

- **An example:**

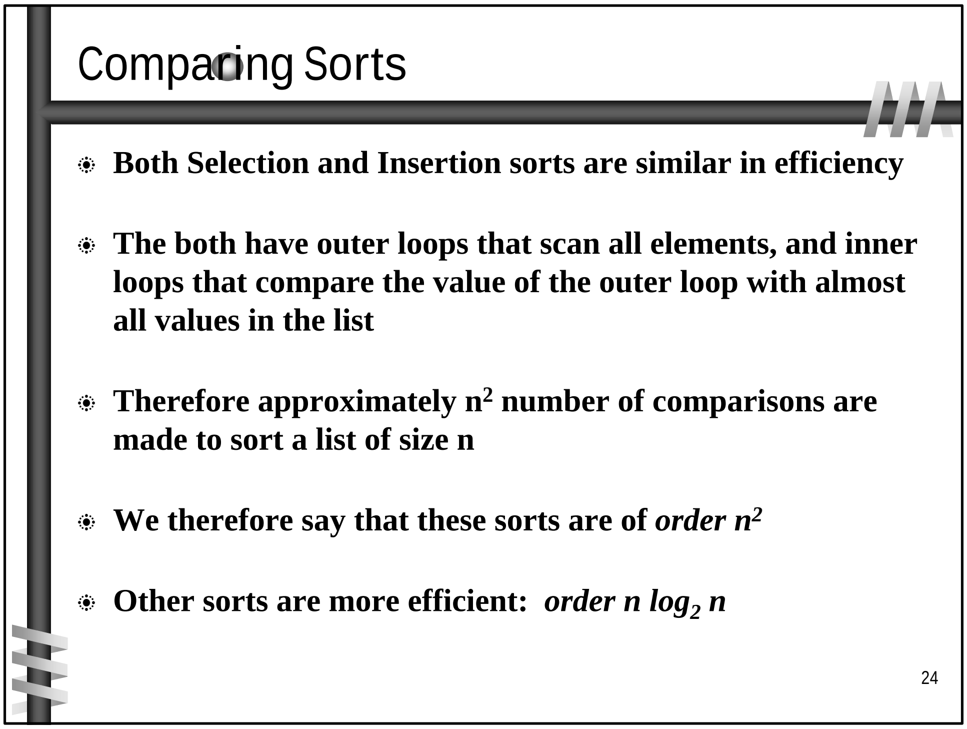
```
original:    3   9   6   1   2
insert 9:    3   9   6   1   2
insert 6:    3   6   9   1   2
insert 1:    1   3   6   9   2
insert 2:    1   2   3   6   9
```

- **See `Sorts.java` (page 290) -- the `insertionSort` method**



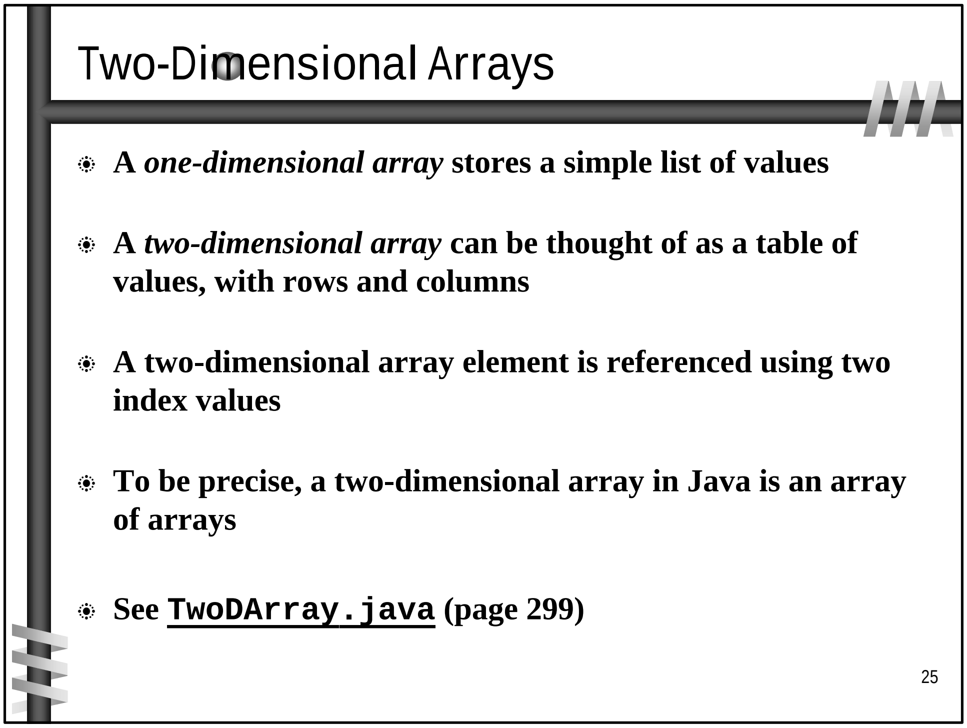
Sorting Objects

- Integers have an inherent order, but the order of a set of objects must be defined by the person defining the class
- Recall that a Java interface can be used as a type name and guarantees that a particular class has implemented particular methods
- We can use the `Comparable` interface to develop a generic sort for a set of objects
- See `SortPhoneList.java (page 294)`
- See `Contact.java` (page 295)
- See `Sorts.java` (page 290)



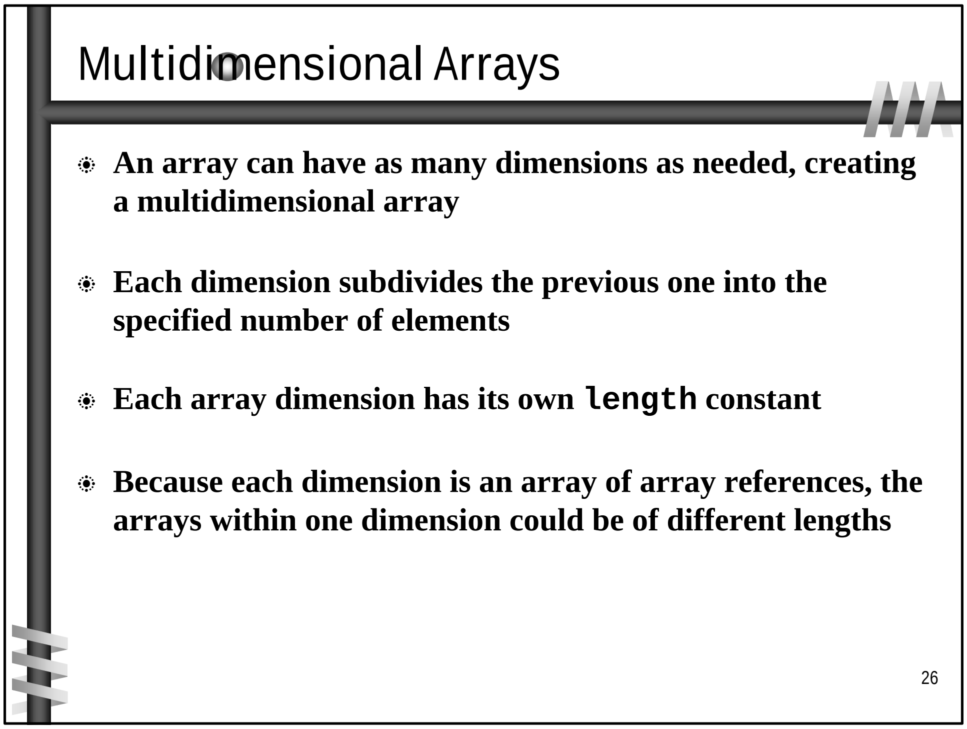
Comparing Sorts

- Both Selection and Insertion sorts are similar in efficiency
- The both have outer loops that scan all elements, and inner loops that compare the value of the outer loop with almost all values in the list
- Therefore approximately n^2 number of comparisons are made to sort a list of size n
- We therefore say that these sorts are of *order* n^2
- Other sorts are more efficient: *order* $n \log_2 n$



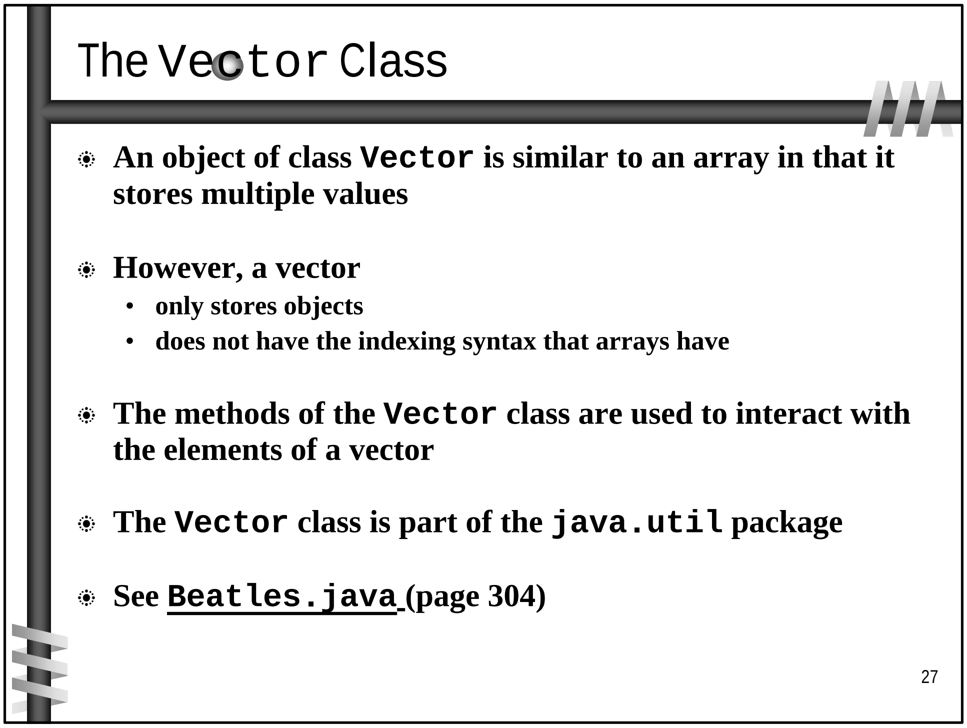
Two-Dimensional Arrays

- A *one-dimensional array* stores a simple list of values
- A *two-dimensional array* can be thought of as a table of values, with rows and columns
- A two-dimensional array element is referenced using two index values
- To be precise, a two-dimensional array in Java is an array of arrays
- See `TwoDArray.java` (page 299)



Multidimensional Arrays

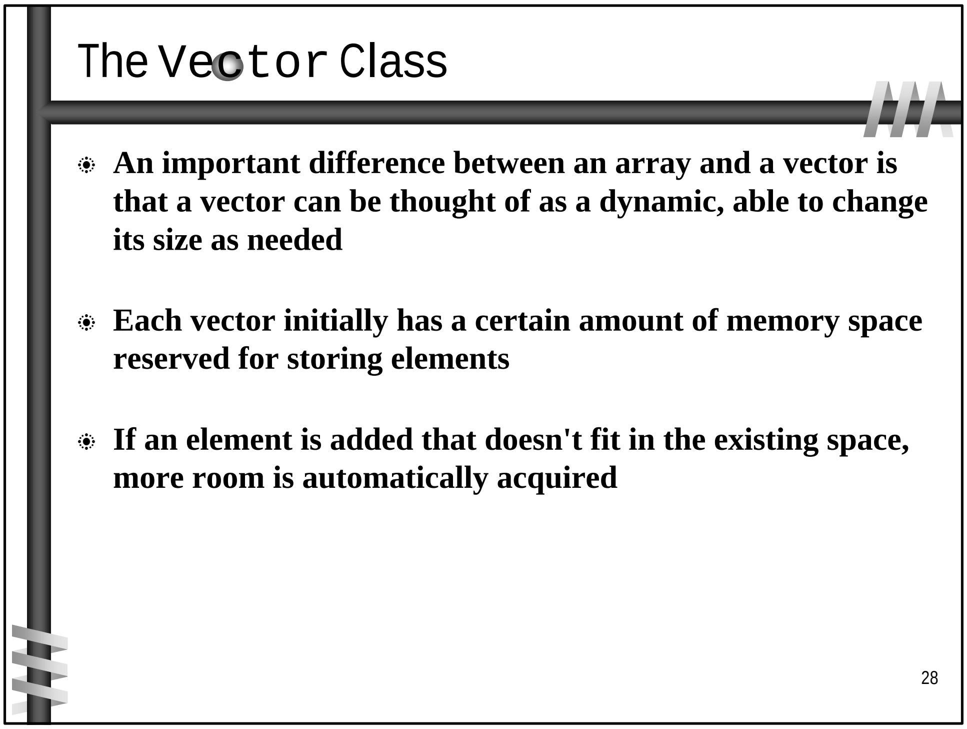
- An array can have as many dimensions as needed, creating a multidimensional array
- Each dimension subdivides the previous one into the specified number of elements
- Each array dimension has its own `length` constant
- Because each dimension is an array of array references, the arrays within one dimension could be of different lengths



The Vector Class

- An object of class **`Vector`** is similar to an array in that it stores multiple values
- However, a vector
  - only stores objects
  - does not have the indexing syntax that arrays have
- The methods of the **`Vector`** class are used to interact with the elements of a vector
- The **`Vector`** class is part of the **`java.util`** package
- See **`Beatles.java`** (page 304)

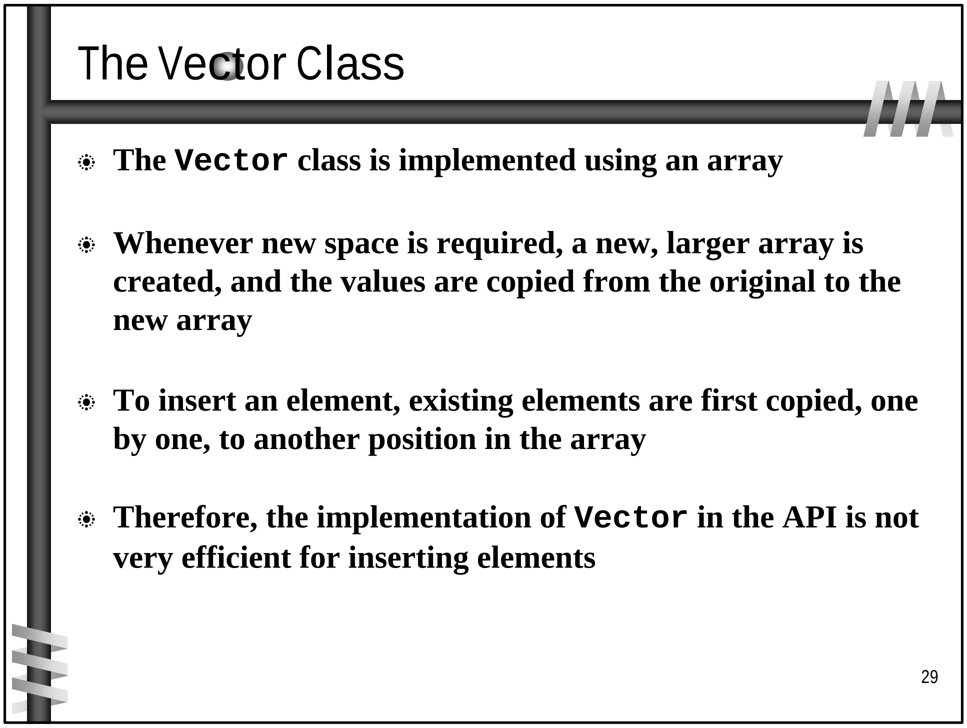
27



The Vector Class

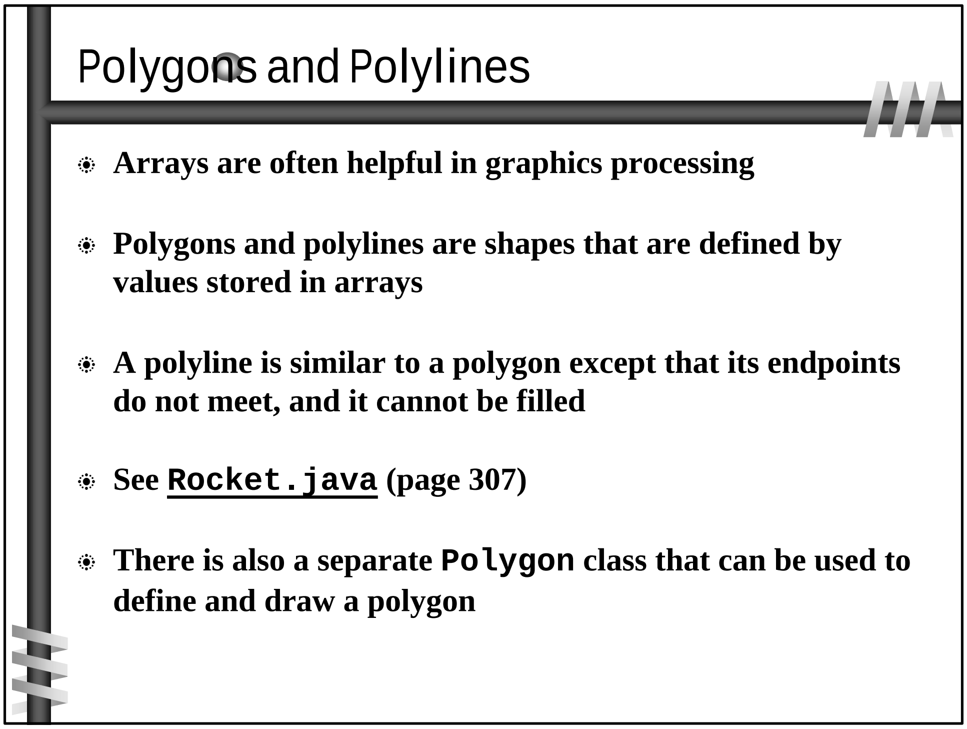
- An important difference between an array and a vector is that a vector can be thought of as a dynamic, able to change its size as needed
- Each vector initially has a certain amount of memory space reserved for storing elements
- If an element is added that doesn't fit in the existing space, more room is automatically acquired

28



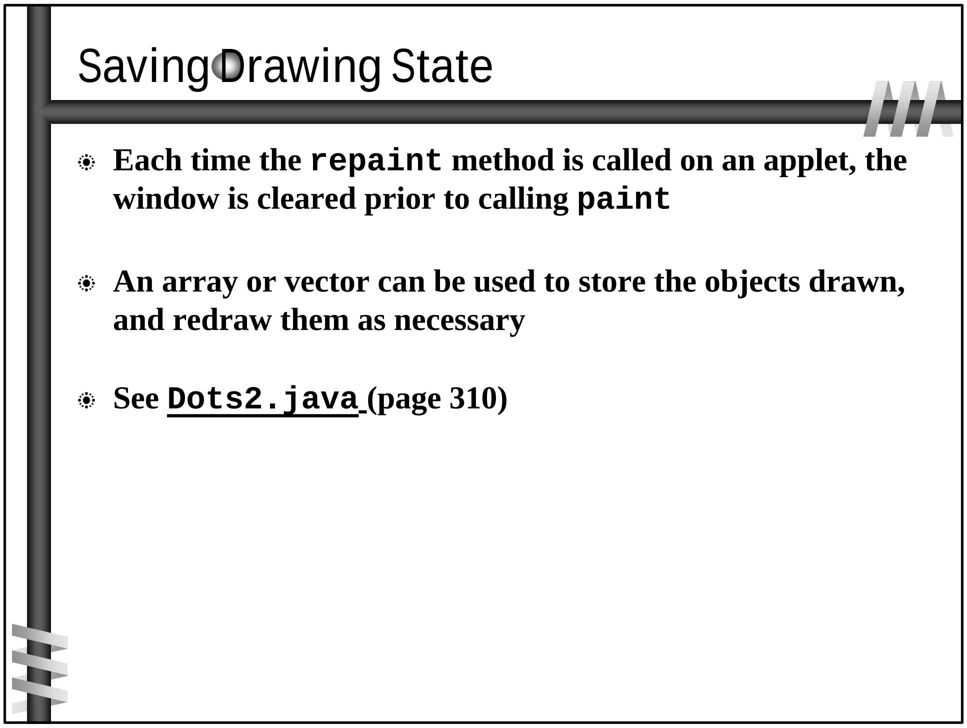
The Vector Class

- The `Vector` class is implemented using an array
- Whenever new space is required, a new, larger array is created, and the values are copied from the original to the new array
- To insert an element, existing elements are first copied, one by one, to another position in the array
- Therefore, the implementation of `Vector` in the API is not very efficient for inserting elements



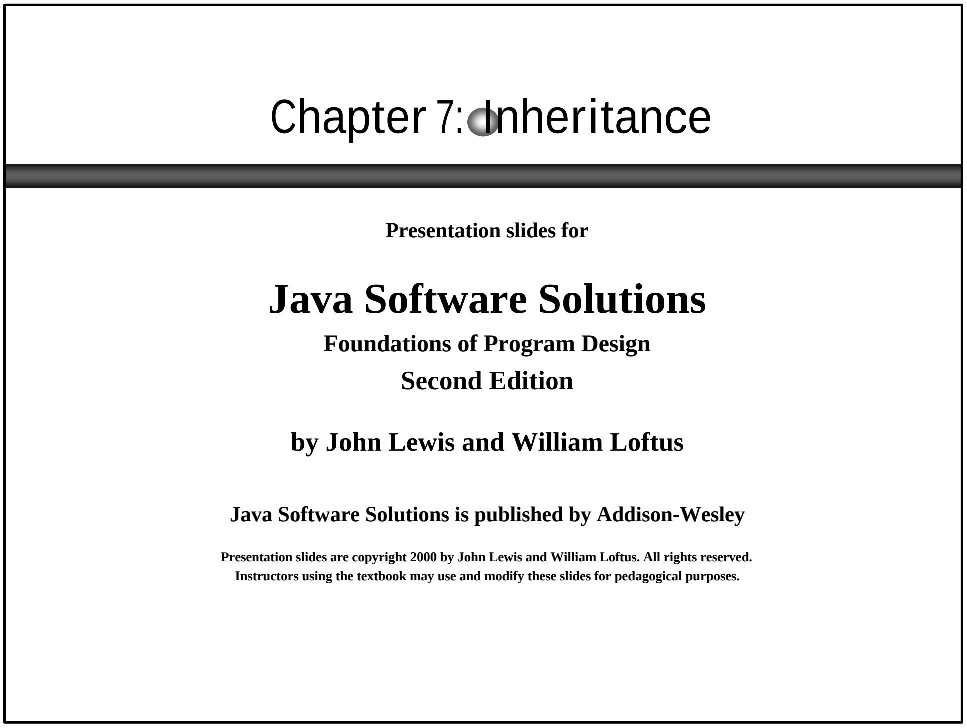
Polygons and Polylines

- Arrays are often helpful in graphics processing
- Polygons and polylines are shapes that are defined by values stored in arrays
- A polyline is similar to a polygon except that its endpoints do not meet, and it cannot be filled
- See `Rocket.java` (page 307)
- There is also a separate `Polygon` class that can be used to define and draw a polygon



Saving Drawing State

- Each time the `repaint` method is called on an applet, the window is cleared prior to calling `paint`
- An array or vector can be used to store the objects drawn, and redraw them as necessary
- See `Dots2.java` (page 310)



Chapter 7: Inheritance

Presentation slides for

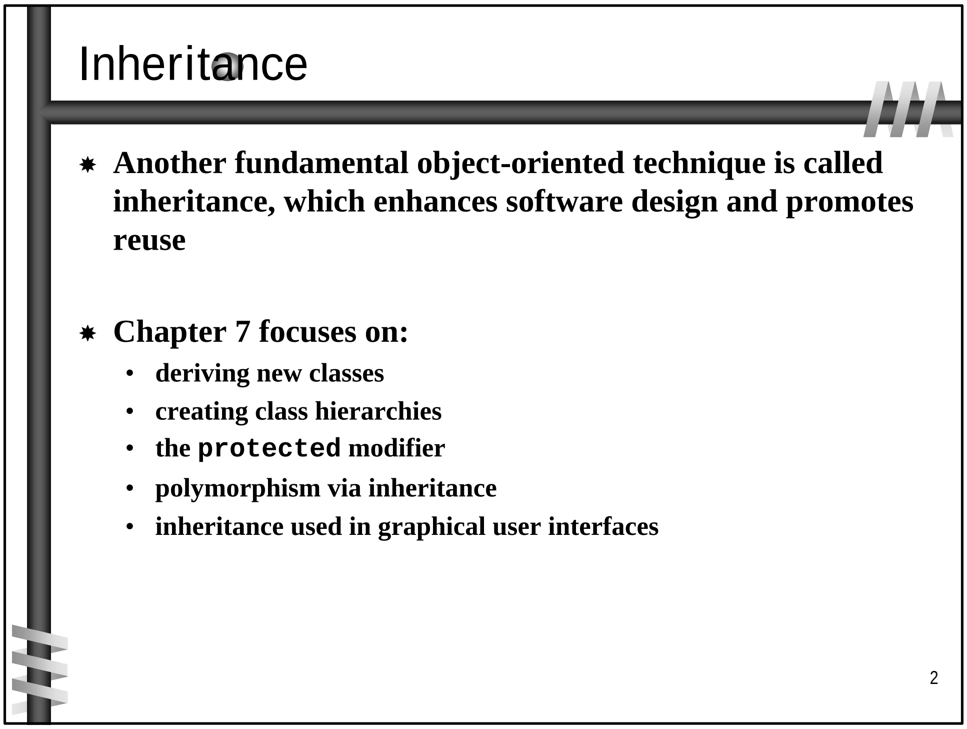
Java Software Solutions

Foundations of Program Design

Second Edition

by John Lewis and William Loftus

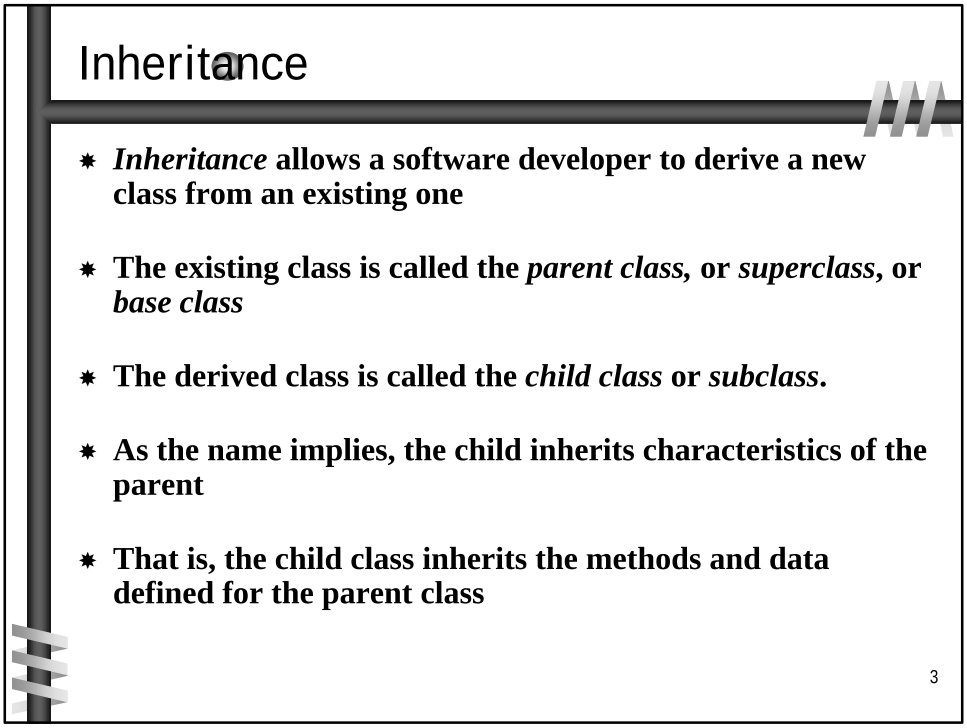
Java Software Solutions is published by Addison-Wesley



Inheritance

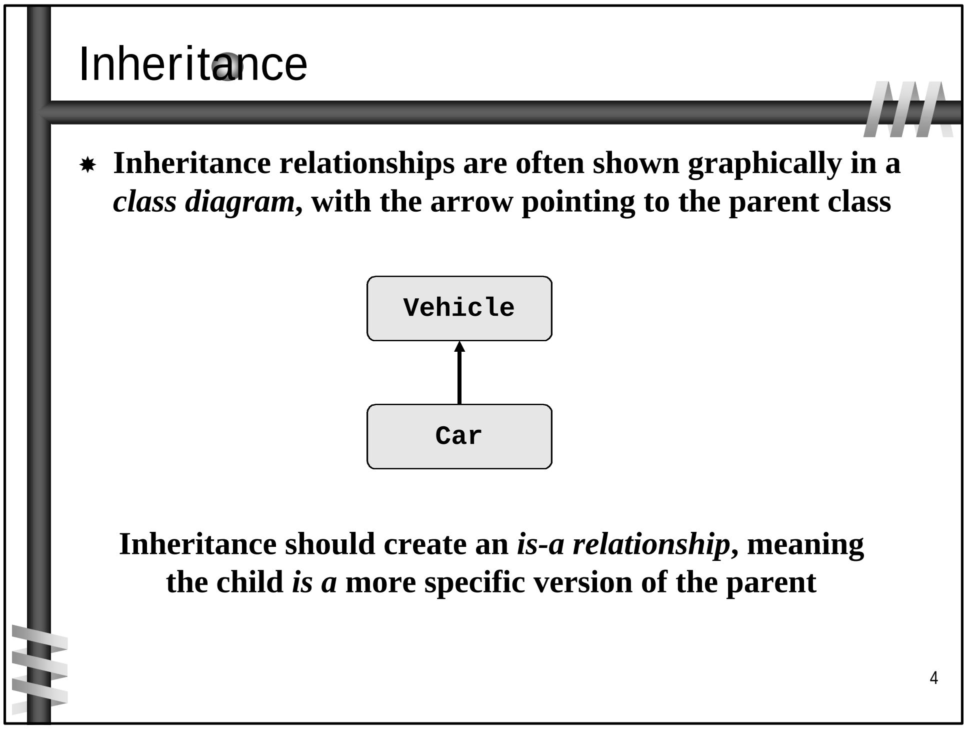
- **Another fundamental object-oriented technique is called inheritance, which enhances software design and promotes reuse**

- **Chapter 7 focuses on:**
  - **deriving new classes**
  - **creating class hierarchies**
  - **the `protected` modifier**
  - **polymorphism via inheritance**
  - **inheritance used in graphical user interfaces**



Inheritance

- *Inheritance* allows a software developer to derive a new class from an existing one
- The existing class is called the *parent class*, or *superclass*, or *base class*
- The derived class is called the *child class* or *subclass*.
- As the name implies, the child inherits characteristics of the parent
- That is, the child class inherits the methods and data defined for the parent class

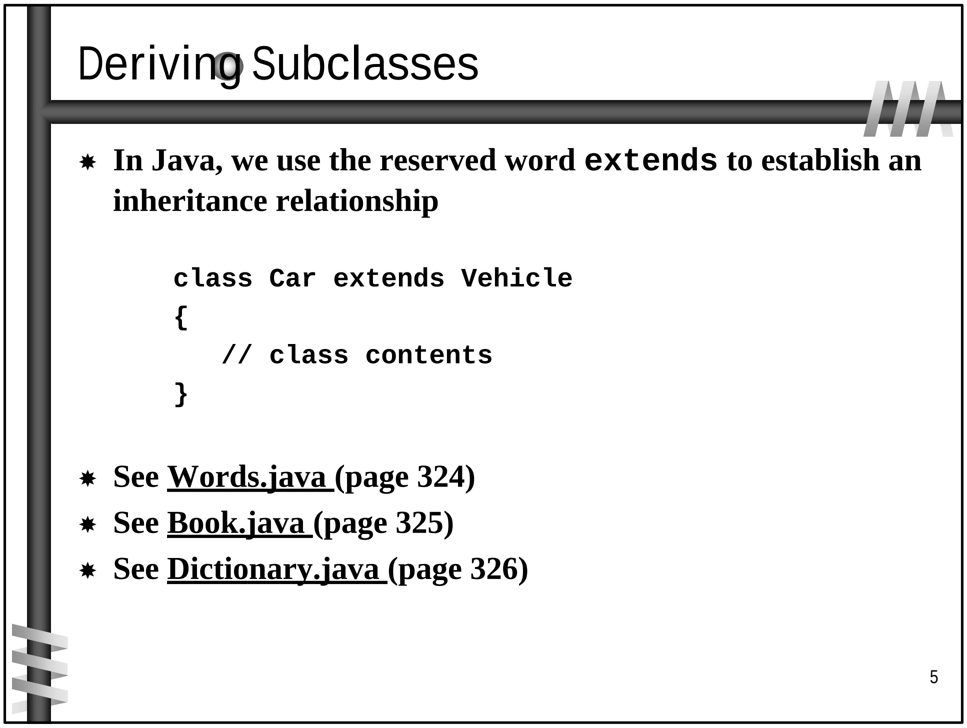


Inheritance

- Inheritance relationships are often shown graphically in a *class diagram*, with the arrow pointing to the parent class

Vehicle
Car

Inheritance should create an *is-a relationship*, meaning the child *is a* more specific version of the parent

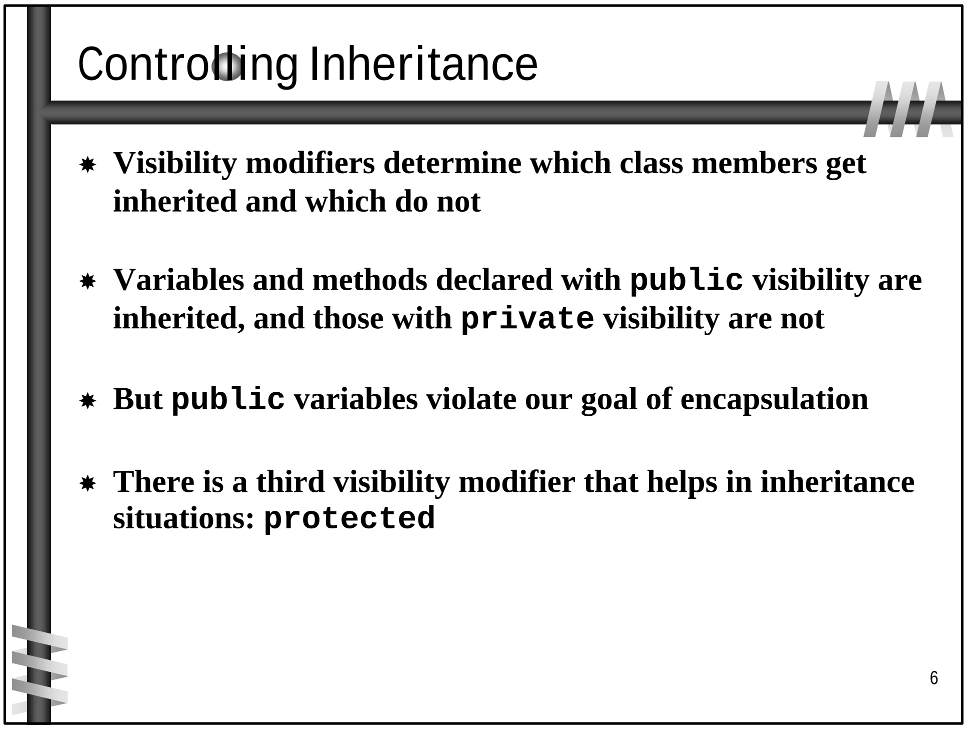


Deriving Subclasses

- In Java, we use the reserved word `extends` to establish an inheritance relationship

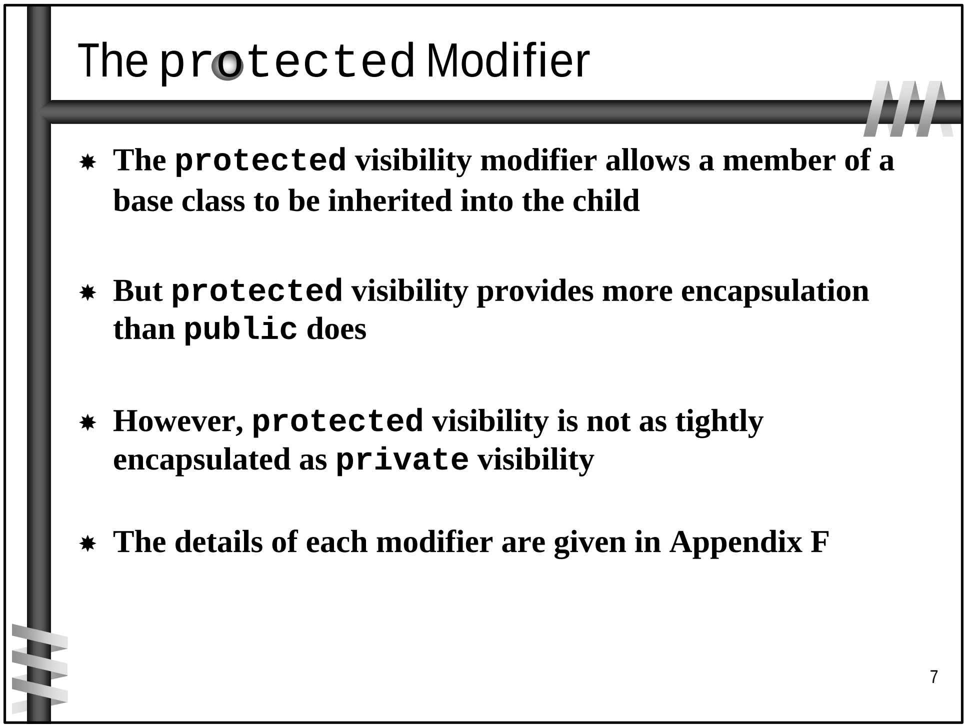
```
class Car extends Vehicle
{
   // class contents
}
```

- See <u>Words.java</u> (page 324)
- See <u>Book.java</u> (page 325)
- See <u>Dictionary.java</u> (page 326)



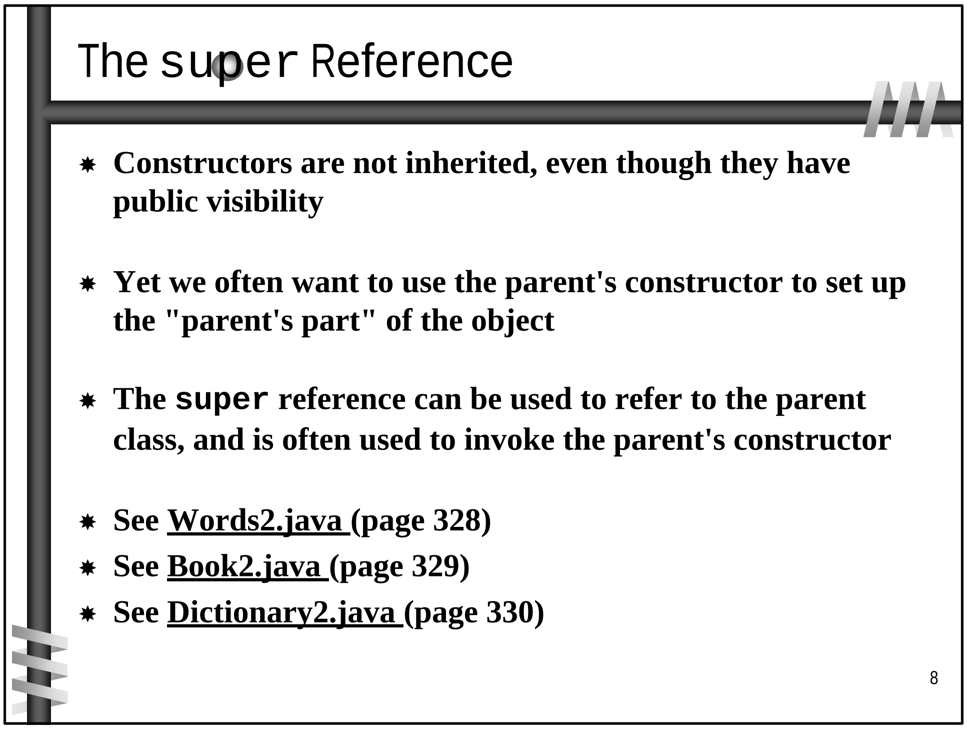
Controlling Inheritance

- Visibility modifiers determine which class members get inherited and which do not
- Variables and methods declared with `public` visibility are inherited, and those with `private` visibility are not
- But `public` variables violate our goal of encapsulation
- There is a third visibility modifier that helps in inheritance situations: `protected`



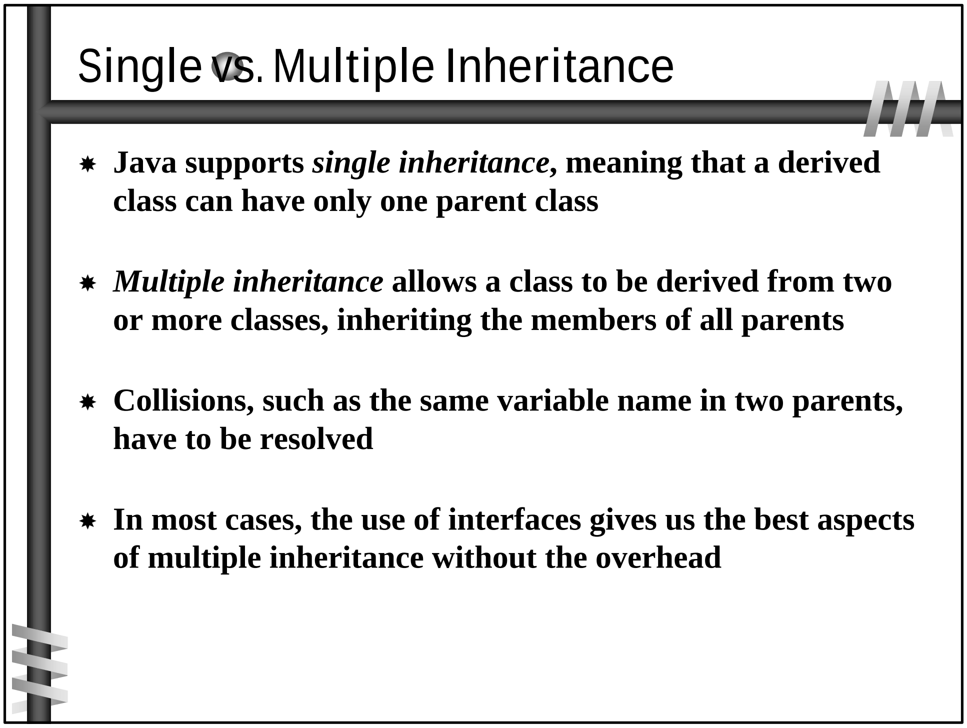
The `protected` Modifier

- The `protected` visibility modifier allows a member of a base class to be inherited into the child
- But `protected` visibility provides more encapsulation than `public` does
- However, `protected` visibility is not as tightly encapsulated as `private` visibility
- The details of each modifier are given in Appendix F



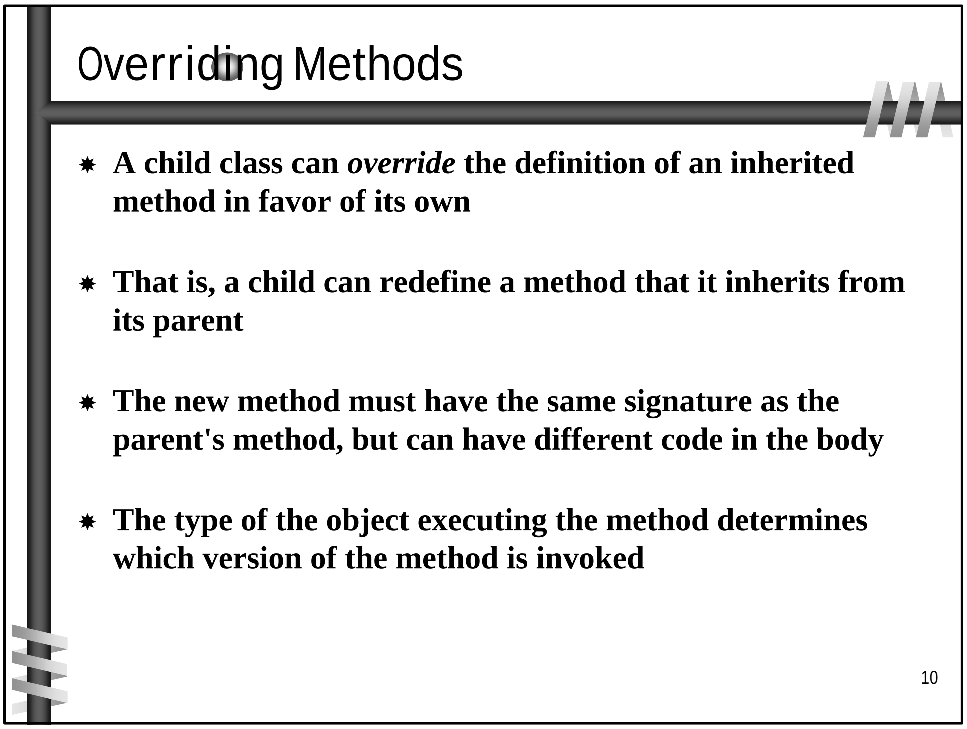
The `super` Reference

- Constructors are not inherited, even though they have public visibility
- Yet we often want to use the parent's constructor to set up the "parent's part" of the object
- The `super` reference can be used to refer to the parent class, and is often used to invoke the parent's constructor
- See <u>Words2.java </u>(page 328)
- See <u>Book2.java </u>(page 329)
- See <u>Dictionary2.java </u>(page 330)



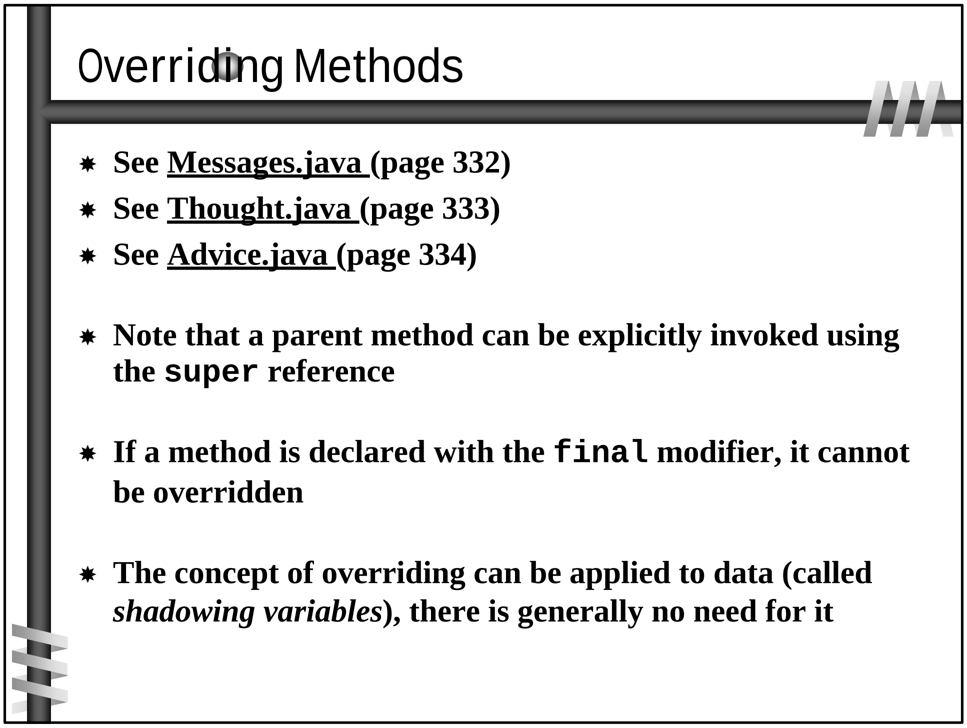
Single vs. Multiple Inheritance

- **Java supports *single inheritance*, meaning that a derived class can have only one parent class**
- ***Multiple inheritance* allows a class to be derived from two or more classes, inheriting the members of all parents**
- **Collisions, such as the same variable name in two parents, have to be resolved**
- **In most cases, the use of interfaces gives us the best aspects of multiple inheritance without the overhead**



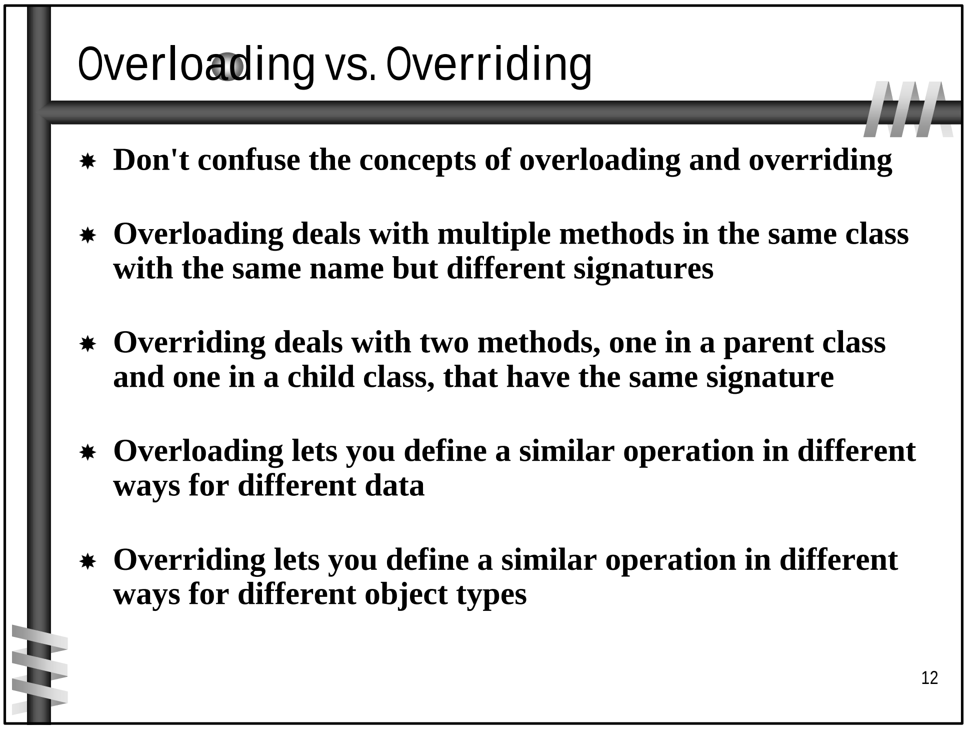
Overriding Methods

- **A child class can *override* the definition of an inherited method in favor of its own**
- **That is, a child can redefine a method that it inherits from its parent**
- **The new method must have the same signature as the parent's method, but can have different code in the body**
- **The type of the object executing the method determines which version of the method is invoked**



Overriding Methods

- See <u>Messages.java</u> (page 332)
- See <u>Thought.java</u> (page 333)
- See <u>Advice.java</u> (page 334)
- Note that a parent method can be explicitly invoked using the `super` reference
- If a method is declared with the `final` modifier, it cannot be overridden
- The concept of overriding can be applied to data (called *shadowing variables*), there is generally no need for it



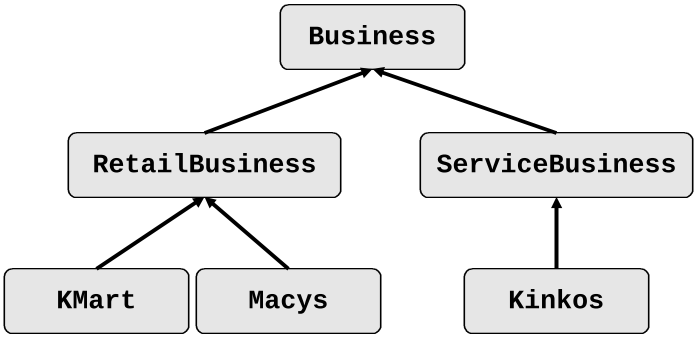
Overloading vs. Overriding

- Don't confuse the concepts of overloading and overriding
- Overloading deals with multiple methods in the same class with the same name but different signatures
- Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature
- Overloading lets you define a similar operation in different ways for different data
- Overriding lets you define a similar operation in different ways for different object types

12

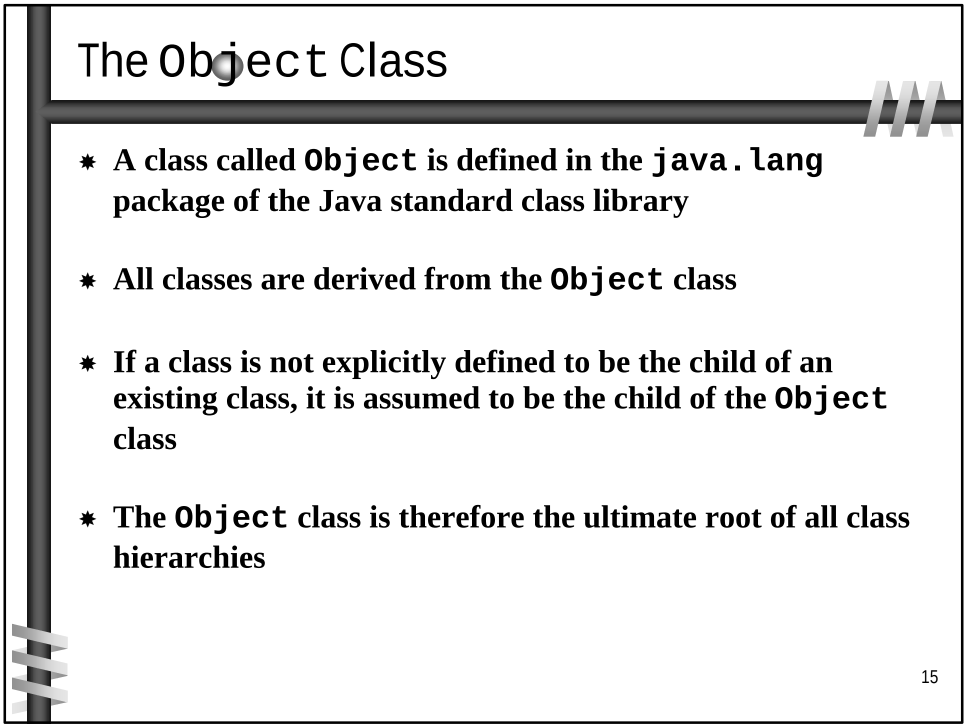
Class Hierarchies

- A child class of one parent can be the parent of another child, forming *class hierarchies*



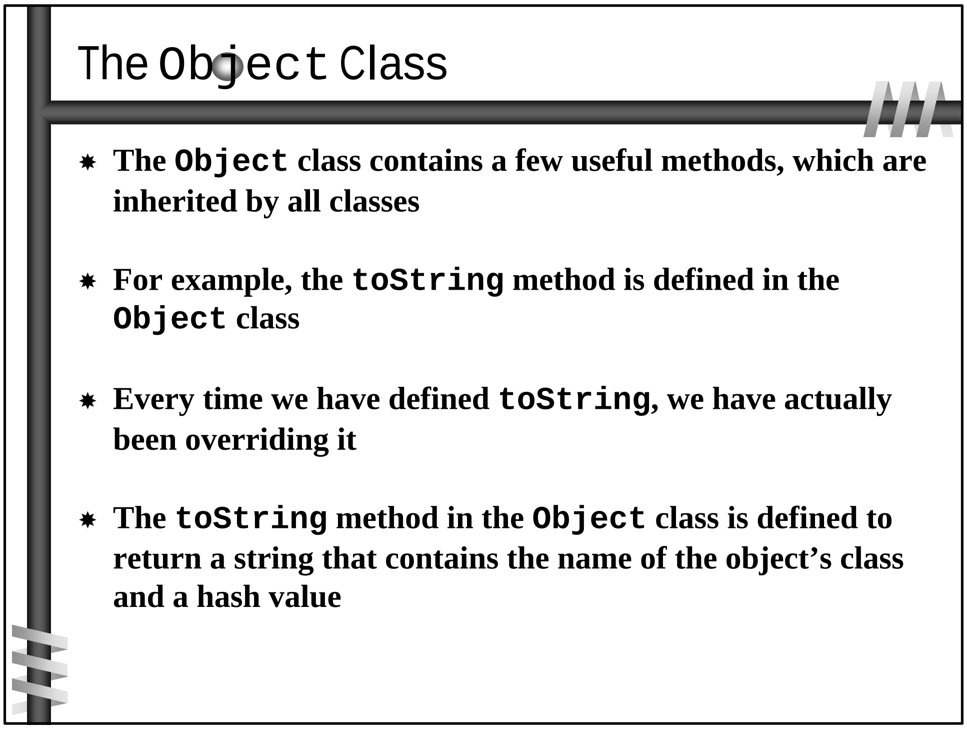
Class Hierarchies

- Two children of the same parent are called *siblings*
- Good class design puts all common features as high in the hierarchy as is reasonable
- An inherited member is continually passed down the line
- Class hierarchies often have to be extended and modified to keep up with changing needs
- There is no single class hierarchy that is appropriate for all situations



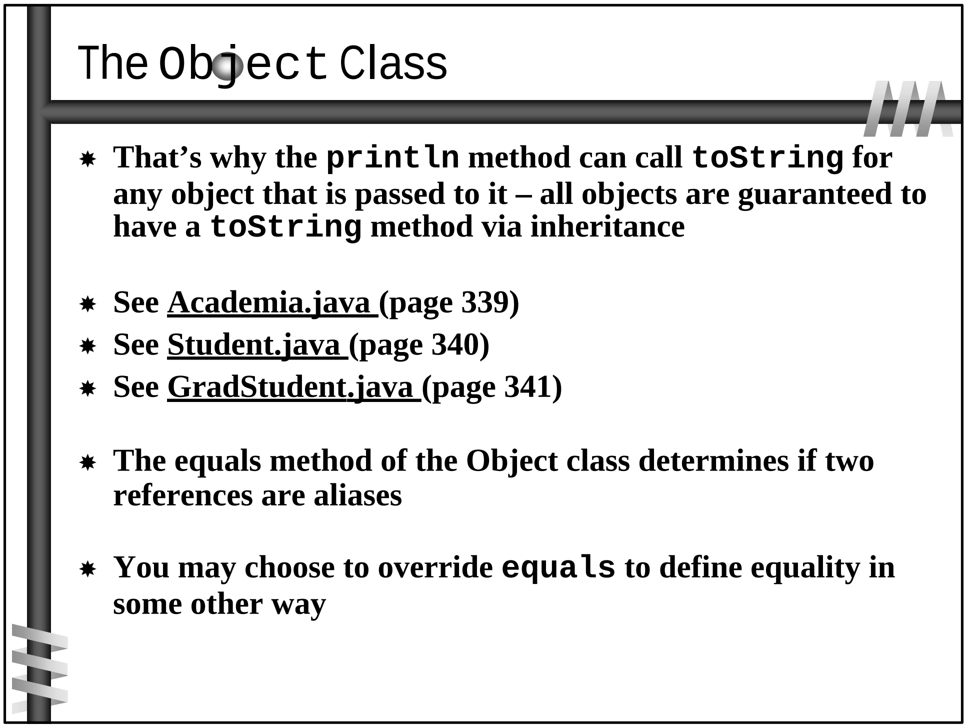
The Object Class

- A class called `Object` is defined in the `java.lang` package of the Java standard class library
- All classes are derived from the `Object` class
- If a class is not explicitly defined to be the child of an existing class, it is assumed to be the child of the `Object` class
- The `Object` class is therefore the ultimate root of all class hierarchies



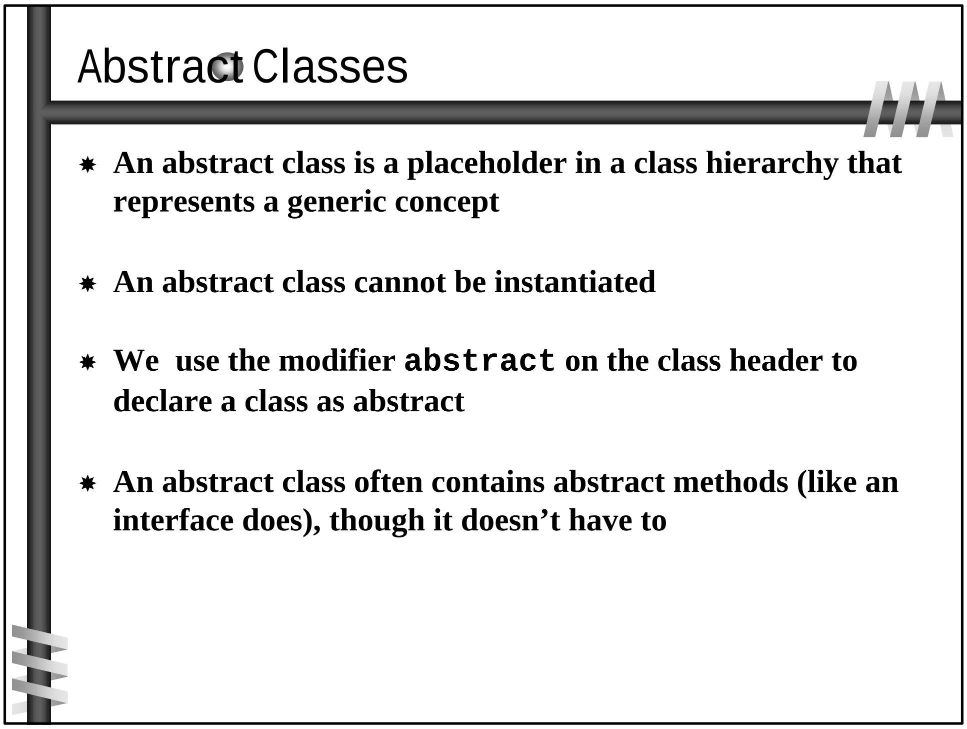
The Object Class

- The `Object` class contains a few useful methods, which are inherited by all classes
- For example, the `toString` method is defined in the `Object` class
- Every time we have defined `toString`, we have actually been overriding it
- The `toString` method in the `Object` class is defined to return a string that contains the name of the object's class and a hash value



The `Object` Class

- That’s why the `println` method can call `toString` for any object that is passed to it – all objects are guaranteed to have a `toString` method via inheritance
- See Academia.java (page 339)
- See Student.java (page 340)
- See GradStudent.java (page 341)
- The equals method of the Object class determines if two references are aliases
- You may choose to override `equals` to define equality in some other way



Abstract Classes

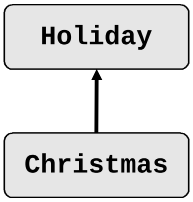
- An abstract class is a placeholder in a class hierarchy that represents a generic concept
- An abstract class cannot be instantiated
- We use the modifier `abstract` on the class header to declare a class as abstract
- An abstract class often contains abstract methods (like an interface does), though it doesn’t have to

Abstract Classes

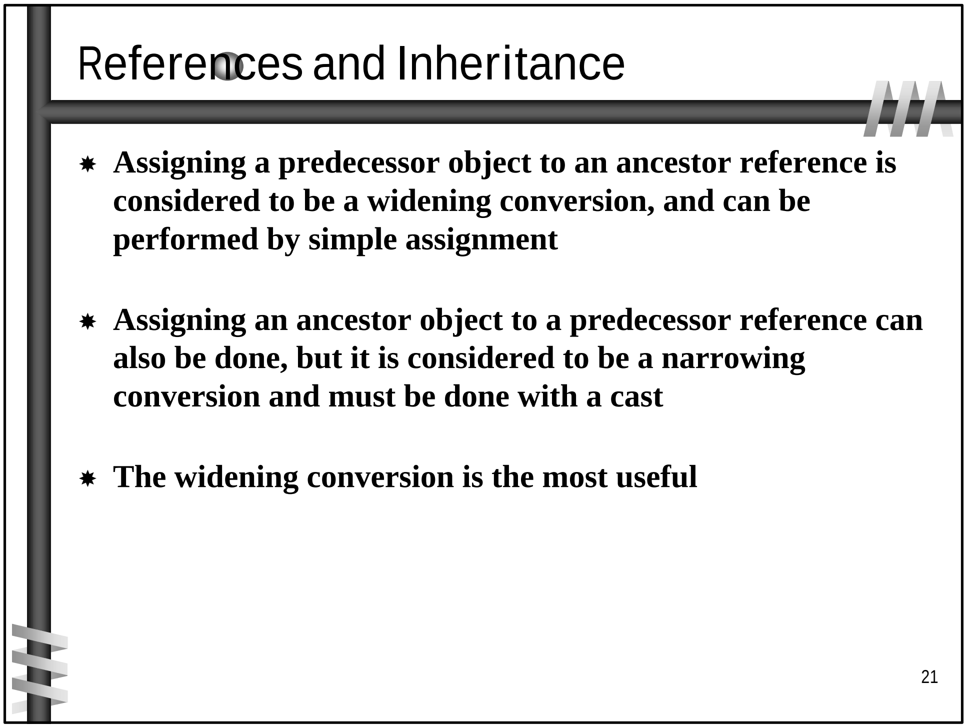
- The child of an abstract class must override the abstract methods of the parent, or it too will be considered abstract
- An abstract method cannot be defined as final (because it must be overridden) or static (because it has no definition yet)
- The use of abstract classes is a design decision; it helps us establish common elements in a class that is to general to instantiate

References and Inheritance

- An object reference can refer to an object of its class, or to an object of any class related to it by inheritance
- For example, if the `Holiday` class is used to derive a child class called `Christmas`, then a `Holiday` reference could actually be used to point to a `Christmas` object

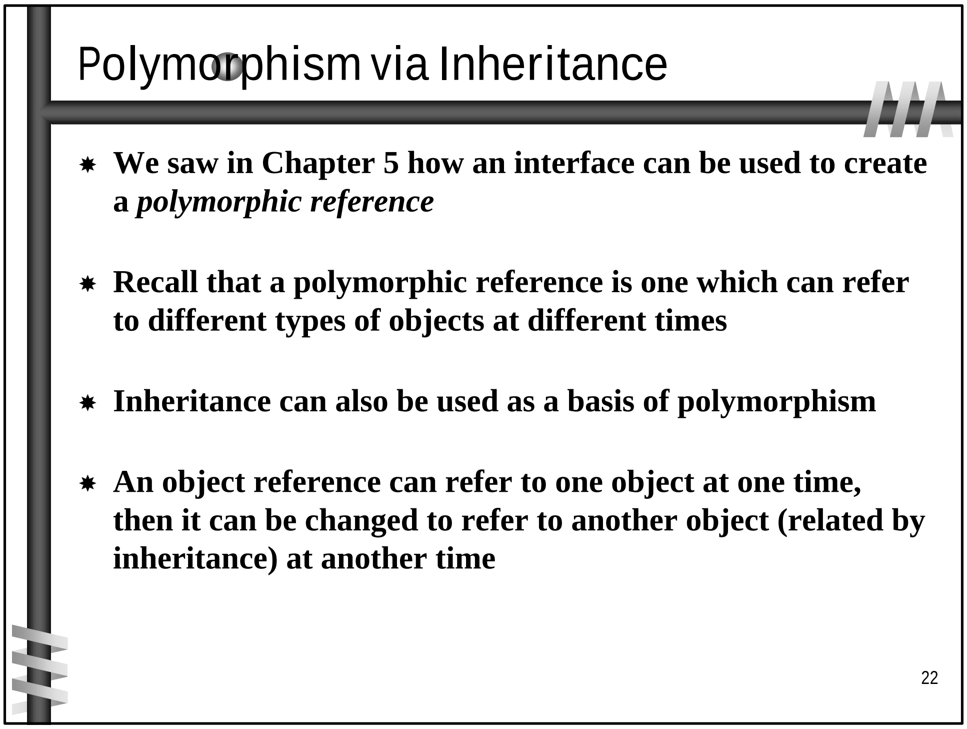


```
Holiday day;
day = new Christmas();
```



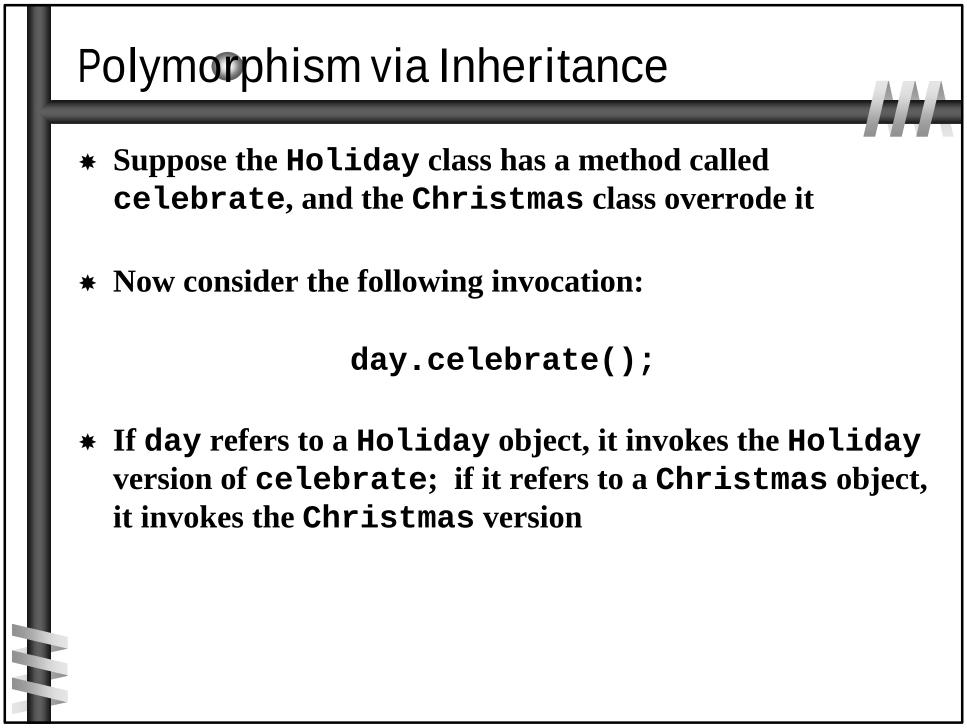
References and Inheritance

- Assigning a predecessor object to an ancestor reference is considered to be a widening conversion, and can be performed by simple assignment
- Assigning an ancestor object to a predecessor reference can also be done, but it is considered to be a narrowing conversion and must be done with a cast
- The widening conversion is the most useful



Polymorphism via Inheritance

- We saw in Chapter 5 how an interface can be used to create a *polymorphic reference*
- Recall that a polymorphic reference is one which can refer to different types of objects at different times
- Inheritance can also be used as a basis of polymorphism
- An object reference can refer to one object at one time, then it can be changed to refer to another object (related by inheritance) at another time



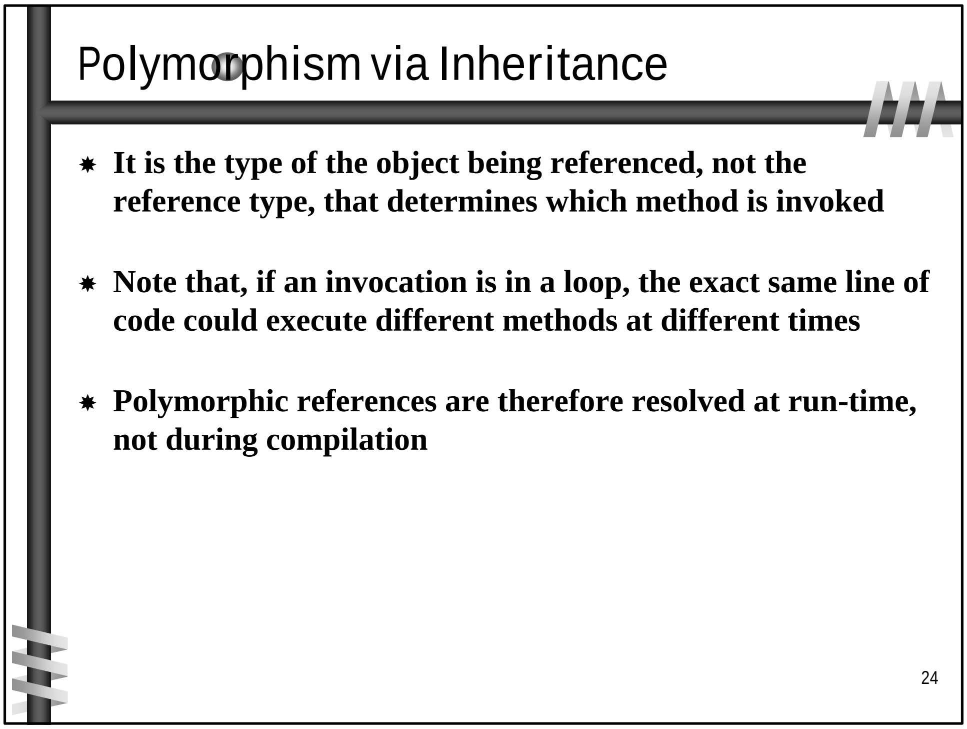
Polymorphism via Inheritance

- **Suppose the `Holiday` class has a method called `celebrate`, and the `Christmas` class overrode it**

- **Now consider the following invocation:**

```
day.celebrate();
```

- **If `day` refers to a `Holiday` object, it invokes the `Holiday` version of `celebrate`; if it refers to a `Christmas` object, it invokes the `Christmas` version**

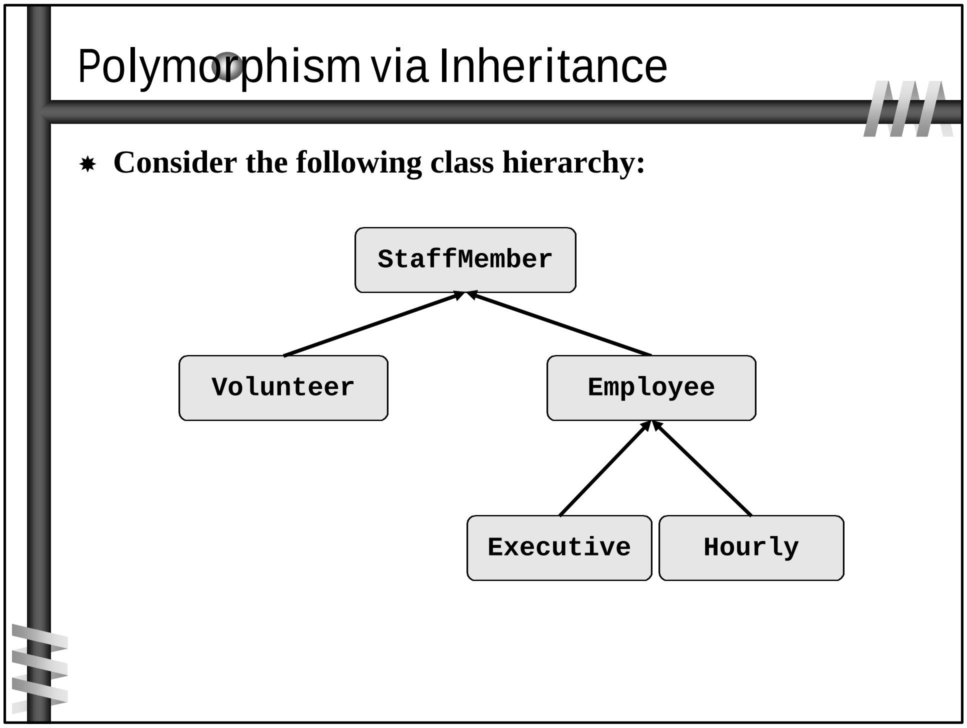


Polymorphism via Inheritance

- **It is the type of the object being referenced, not the reference type, that determines which method is invoked**

- **Note that, if an invocation is in a loop, the exact same line of code could execute different methods at different times**

- **Polymorphic references are therefore resolved at run-time, not during compilation**



Polymorphism via Inheritance

- Consider the following class hierarchy:

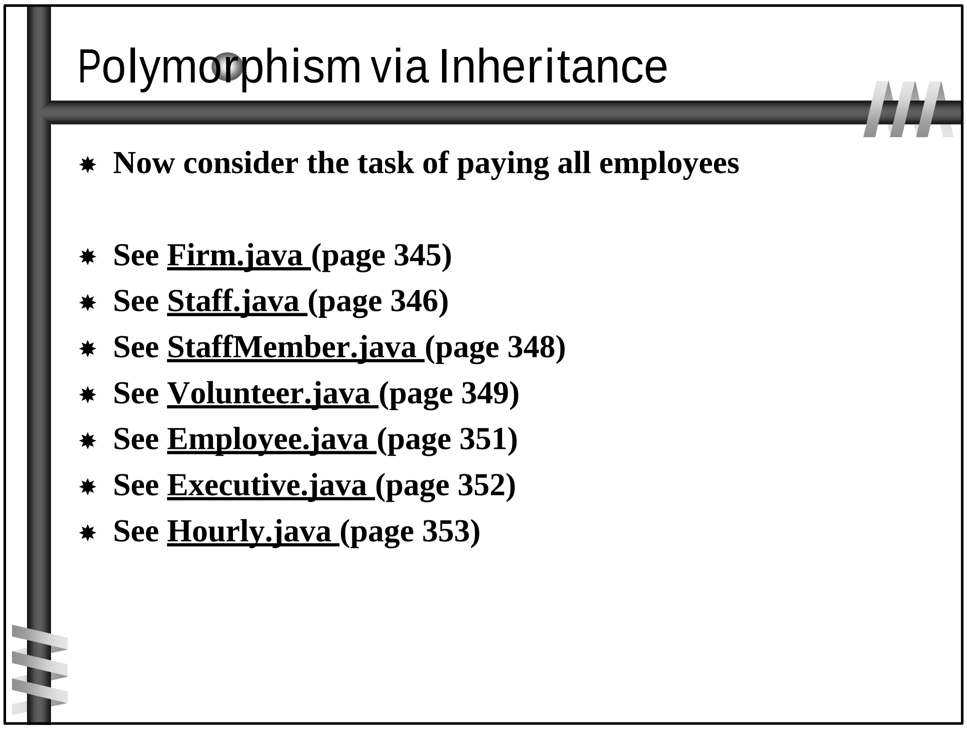
StaffMember

Volunteer

Employee

Executive

Hourly



Polymorphism via Inheritance

- Now consider the task of paying all employees

- See Firm.java (page 345)
- See Staff.java (page 346)
- See StaffMember.java (page 348)
- See Volunteer.java (page 349)
- See Employee.java (page 351)
- See Executive.java (page 352)
- See Hourly.java (page 353)

Indirect Access

- An inherited member can be referenced directly by name in the child class, as if it were declared in the child class
- But even if a method or variable is not inherited by a child, it can still be accessed indirectly through parent methods
- See FoodAnalysis.java (page 355)
- See FoodItem.java (page 356)
- See Pizza.java (page 357)

Interface Hierarchies

- Inheritance can be applied to interfaces as well as classes
- One interface can be used as the parent of another
- The child interface inherits all abstract methods of the parent
- A class implementing the child interface must define all methods from both the parent and child interfaces
- Note that class hierarchies and interface hierarchies are distinct (the do not overlap)

Applets and Inheritance

- An applet is an excellent example of inheritance
- Recall that when we define an applet, we extend the `Applet` class
- The `Applet` class already handles all the details about applet creation and execution, including the interaction with a web browser
- Our applet classes only have to deal with issues that specifically relate to what our particular applet will do

Extending Event Adapter Classes

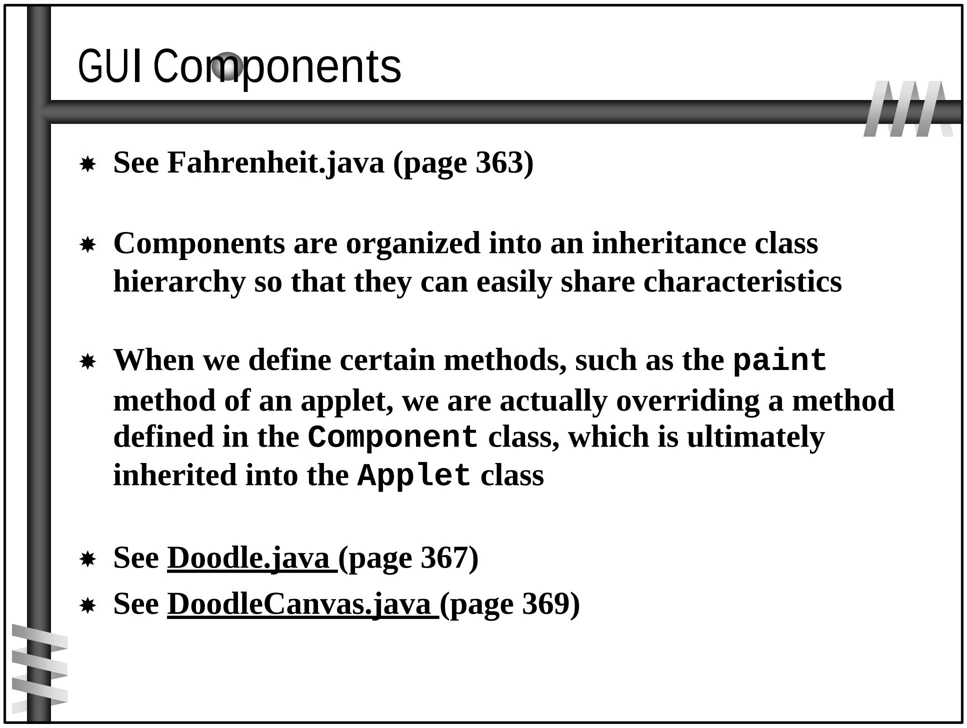
- In Chapter 5 we discussed the creation of listener classes by implementing a particular interface (such as `MouseListener` interface)
- A listener can also be created by extending a special *adapter class* of the Java class library
- Each listener interface has a corresponding adapter class (such as the `MouseAdapter` class)
- Each adapter class implements the corresponding listener and provides empty method definitions

Extending Event Adapter Classes

- When you derive a listener class from an adapter class, you override any event methods of interest (such as the `mouseClicked` method)
- Note that this avoids the need to create empty definitions for unused events
- See OffCenter.java (page 360)

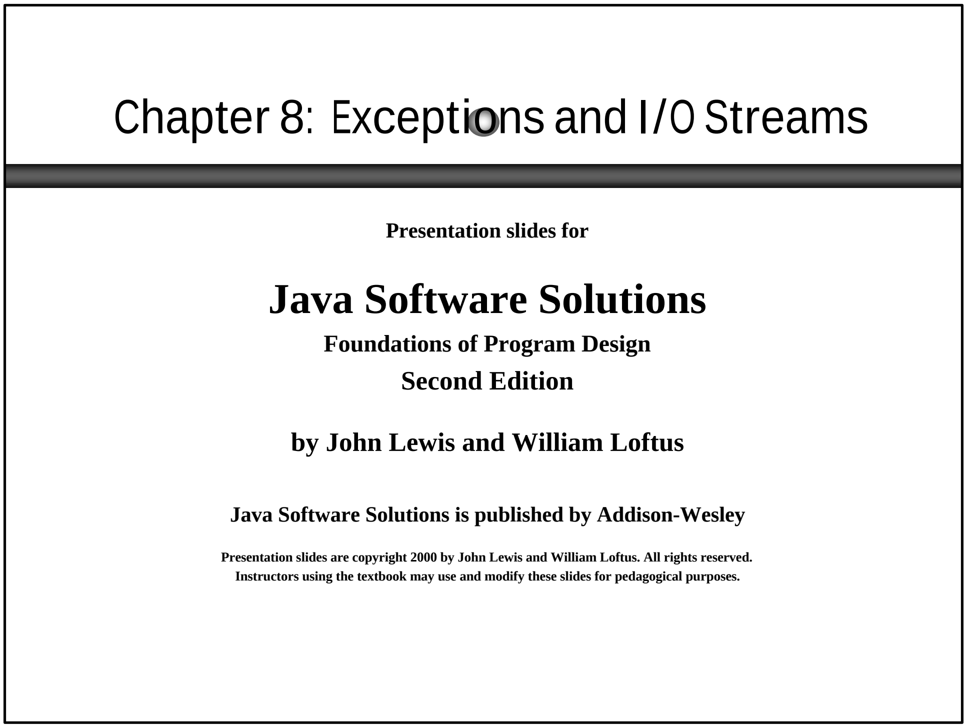
GUI Components

- A *GUI component* is an object that represents a visual entity in an graphical user interface (such as a button or slider)
- Components can generate events to which listener objects can respond
- For example, an applet is a component that can generate mouse events
- An applet is also a special kind of component, called a *container*, in which other components can be placed



GUI Components

- See Fahrenheit.java (page 363)
- Components are organized into an inheritance class hierarchy so that they can easily share characteristics
- When we define certain methods, such as the `paint` method of an applet, we are actually overriding a method defined in the `Component` class, which is ultimately inherited into the `Applet` class
- See Doodle.java (page 367)
- See DoodleCanvas.java (page 369)



Chapter 8: Exceptions and I/O Streams

Presentation slides for

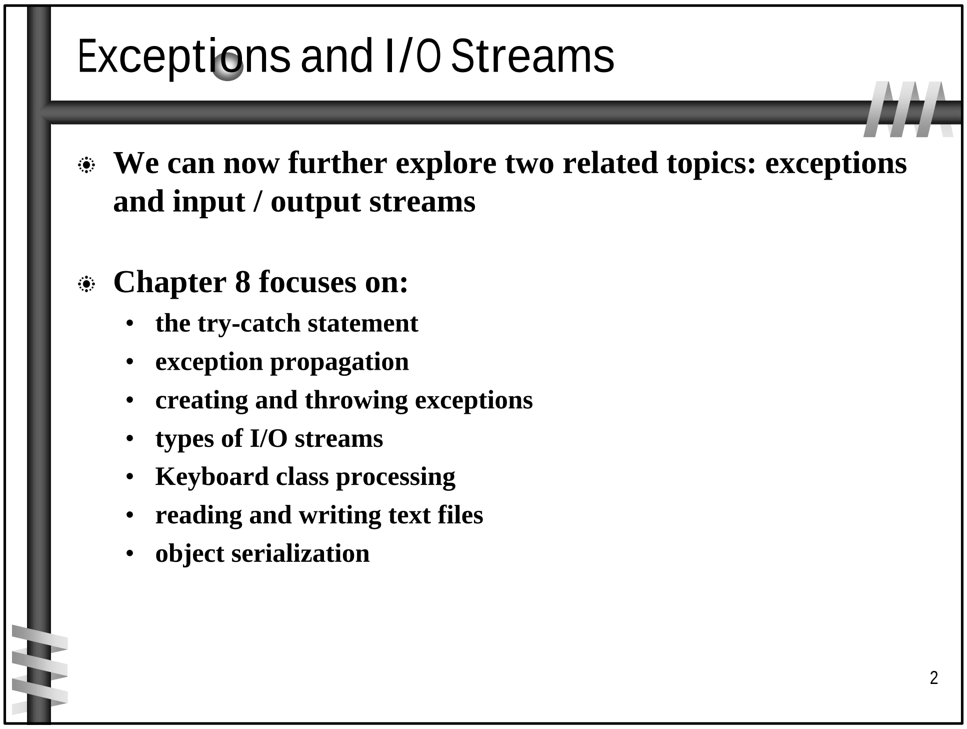
Java Software Solutions

Foundations of Program Design

Second Edition

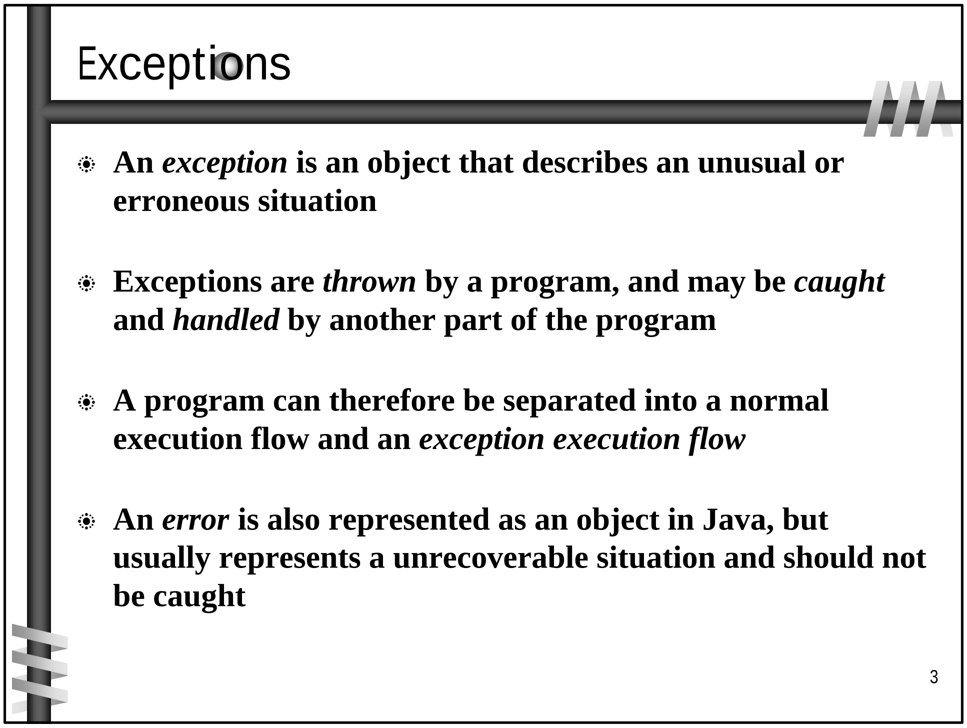
by John Lewis and William Loftus

Java Software Solutions is published by Addison-Wesley



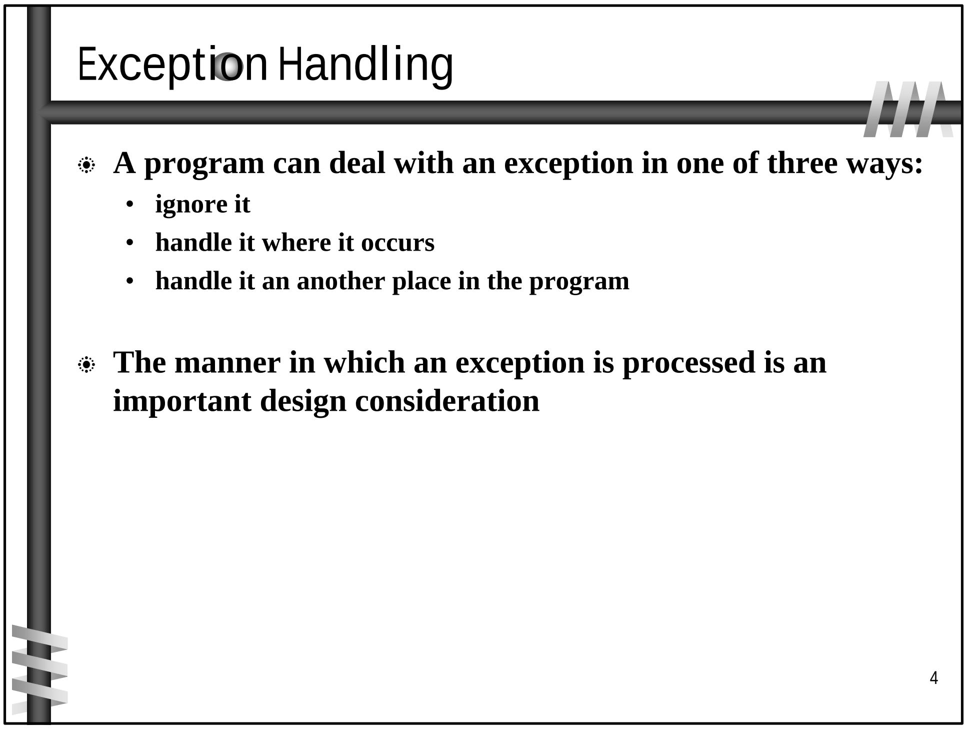
Exceptions and I/O Streams

- **We can now further explore two related topics: exceptions and input / output streams**
- **Chapter 8 focuses on:**
  - **the try-catch statement**
  - **exception propagation**
  - **creating and throwing exceptions**
  - **types of I/O streams**
  - **Keyboard class processing**
  - **reading and writing text files**
  - **object serialization**



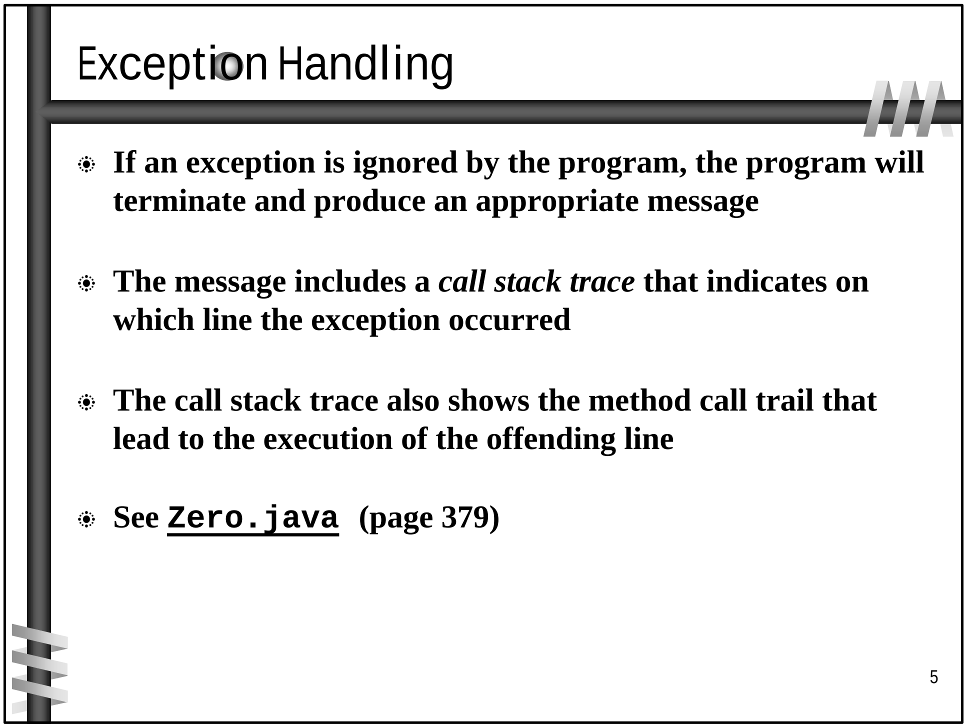
Exceptions

- An *exception* is an object that describes an unusual or erroneous situation
- Exceptions are *thrown* by a program, and may be *caught* and *handled* by another part of the program
- A program can therefore be separated into a normal execution flow and an *exception execution flow*
- An *error* is also represented as an object in Java, but usually represents a unrecoverable situation and should not be caught



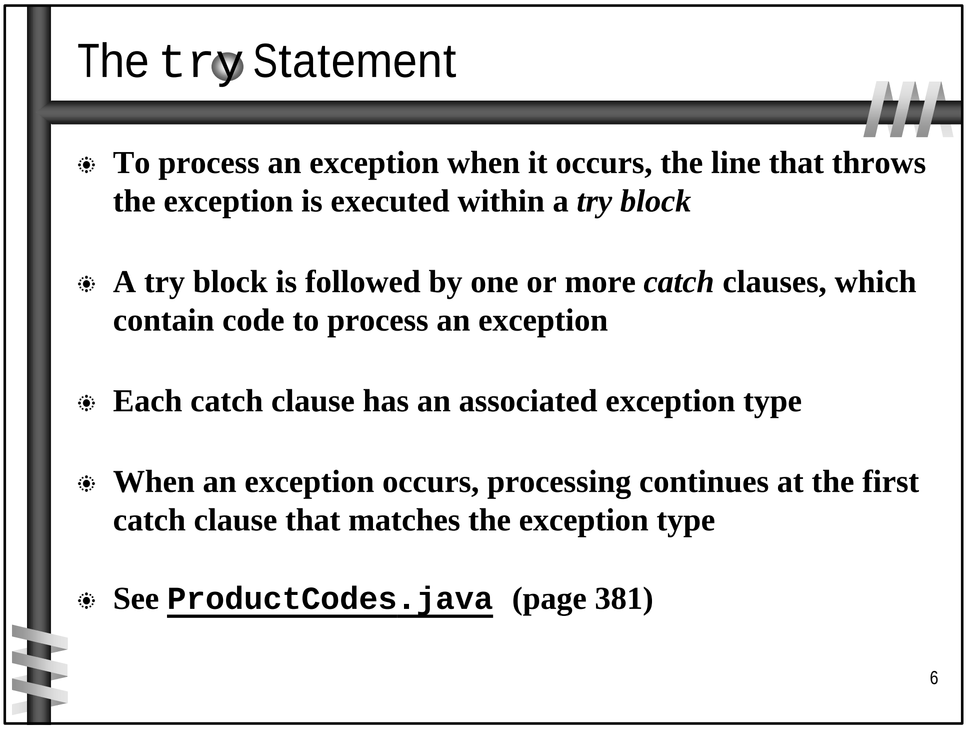
Exception Handling

- A program can deal with an exception in one of three ways:
  - ignore it
  - handle it where it occurs
  - handle it an another place in the program
- The manner in which an exception is processed is an important design consideration



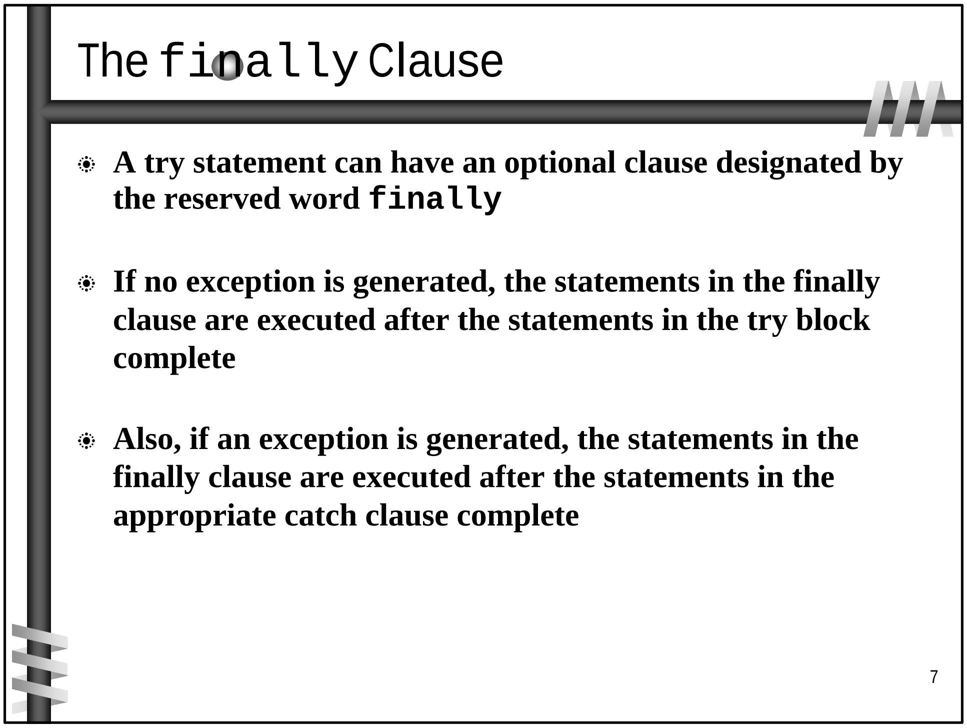
Exception Handling

- If an exception is ignored by the program, the program will terminate and produce an appropriate message
- The message includes a *call stack trace* that indicates on which line the exception occurred
- The call stack trace also shows the method call trail that lead to the execution of the offending line
- See `Zero.java` (page 379)



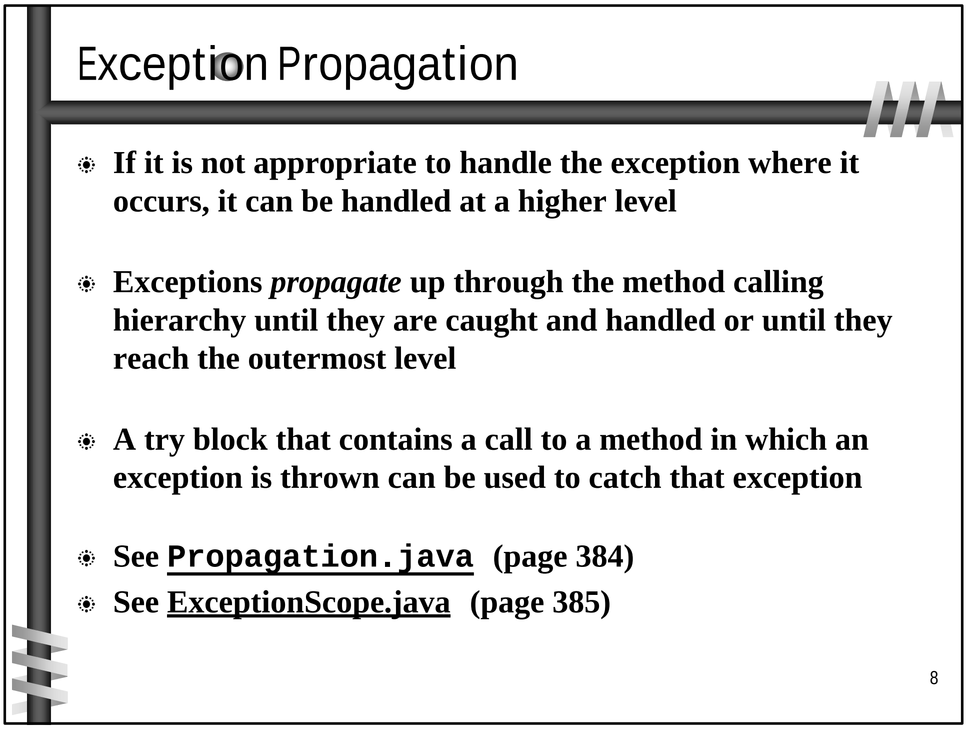
The `try` Statement

- To process an exception when it occurs, the line that throws the exception is executed within a *try block*
- A try block is followed by one or more *catch* clauses, which contain code to process an exception
- Each catch clause has an associated exception type
- When an exception occurs, processing continues at the first catch clause that matches the exception type
- See `ProductCodes.java` (page 381)



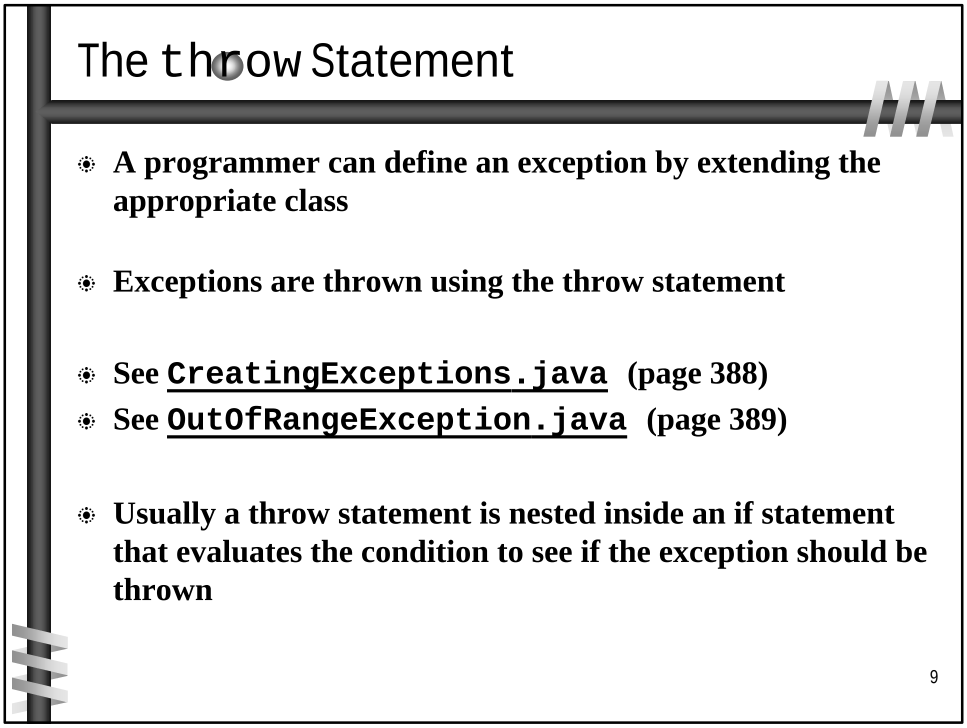
The `finally` Clause

- **A try statement can have an optional clause designated by the reserved word `finally`**
- **If no exception is generated, the statements in the finally clause are executed after the statements in the try block complete**
- **Also, if an exception is generated, the statements in the finally clause are executed after the statements in the appropriate catch clause complete**



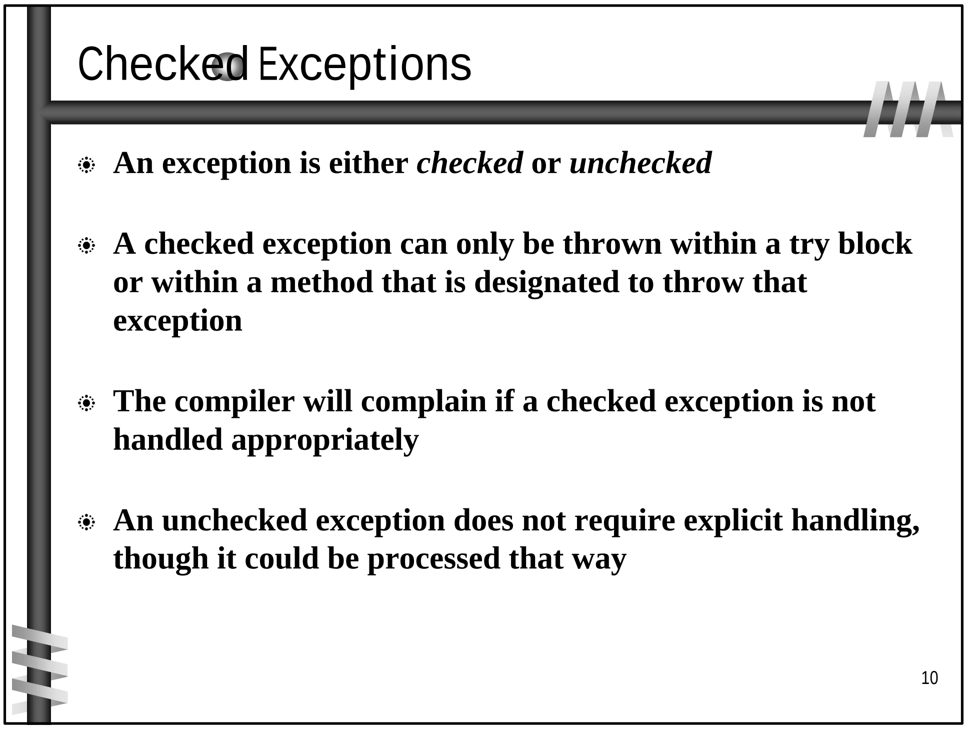
Exception Propagation

- **If it is not appropriate to handle the exception where it occurs, it can be handled at a higher level**
- **Exceptions *propagate* up through the method calling hierarchy until they are caught and handled or until they reach the outermost level**
- **A try block that contains a call to a method in which an exception is thrown can be used to catch that exception**
- **See `Propagation.java` (page 384)**
- **See ExceptionScope.java (page 385)**



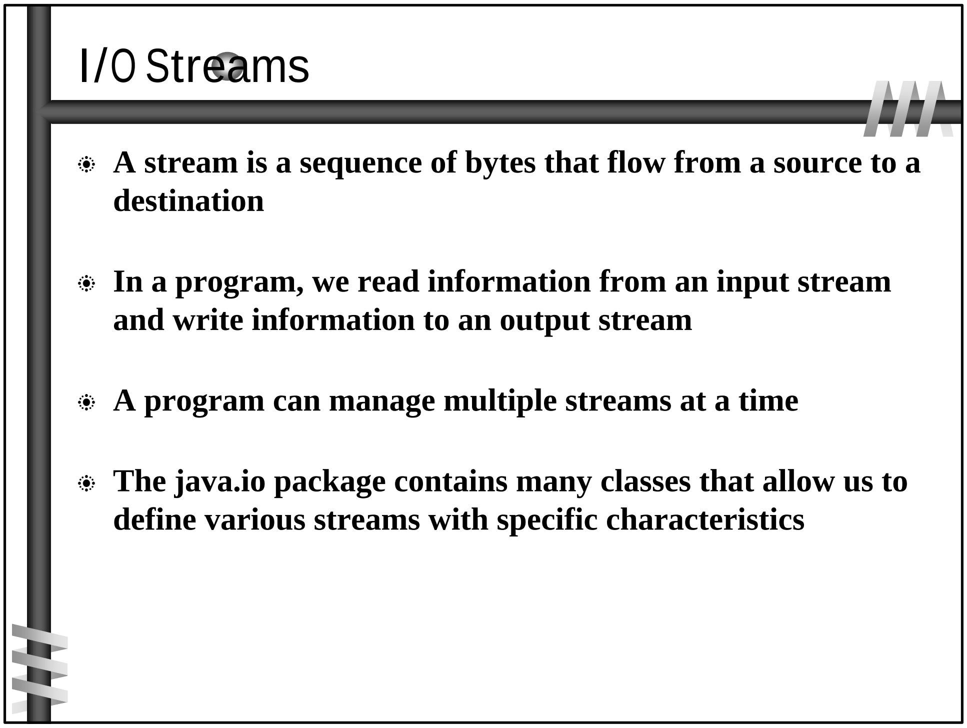
The `throw` Statement

- A programmer can define an exception by extending the appropriate class
- Exceptions are thrown using the throw statement
- See `CreatingExceptions.java` (page 388)
- See `OutOfRangeException.java` (page 389)
- Usually a throw statement is nested inside an if statement that evaluates the condition to see if the exception should be thrown



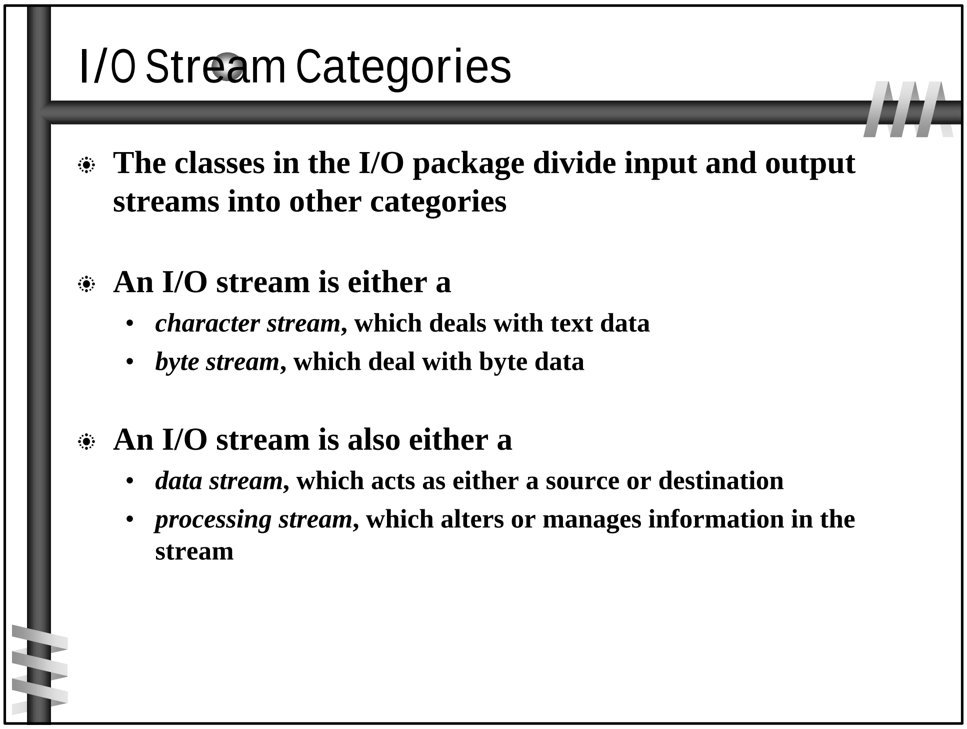
Checked Exceptions

- An exception is either *checked* or *unchecked*
- A checked exception can only be thrown within a try block or within a method that is designated to throw that exception
- The compiler will complain if a checked exception is not handled appropriately
- An unchecked exception does not require explicit handling, though it could be processed that way



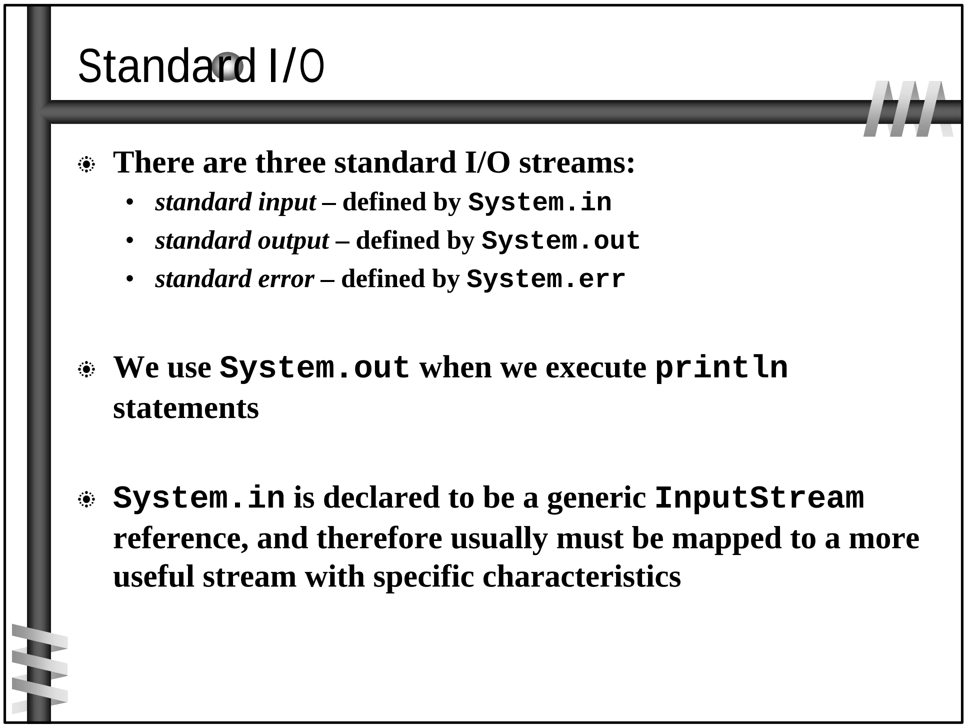
I/O Streams

- **A stream is a sequence of bytes that flow from a source to a destination**
- **In a program, we read information from an input stream and write information to an output stream**
- **A program can manage multiple streams at a time**
- **The java.io package contains many classes that allow us to define various streams with specific characteristics**



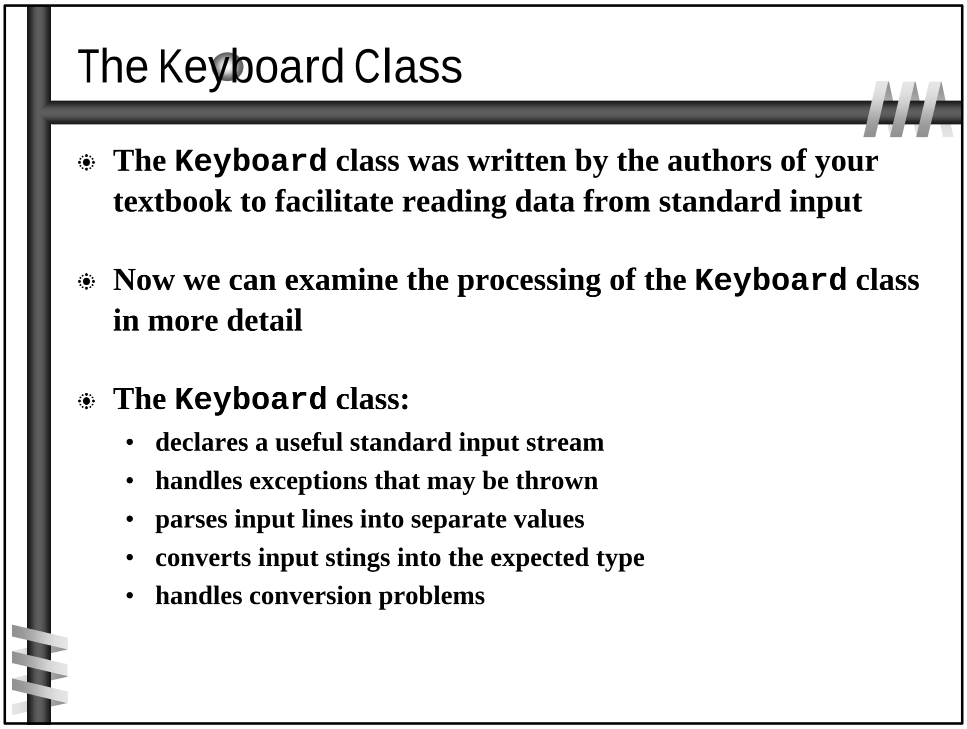
I/O Stream Categories

- **The classes in the I/O package divide input and output streams into other categories**
- **An I/O stream is either a**
  - ***character stream*, which deals with text data**
  - ***byte stream*, which deal with byte data**
- **An I/O stream is also either a**
  - ***data stream*, which acts as either a source or destination**
  - ***processing stream*, which alters or manages information in the stream**



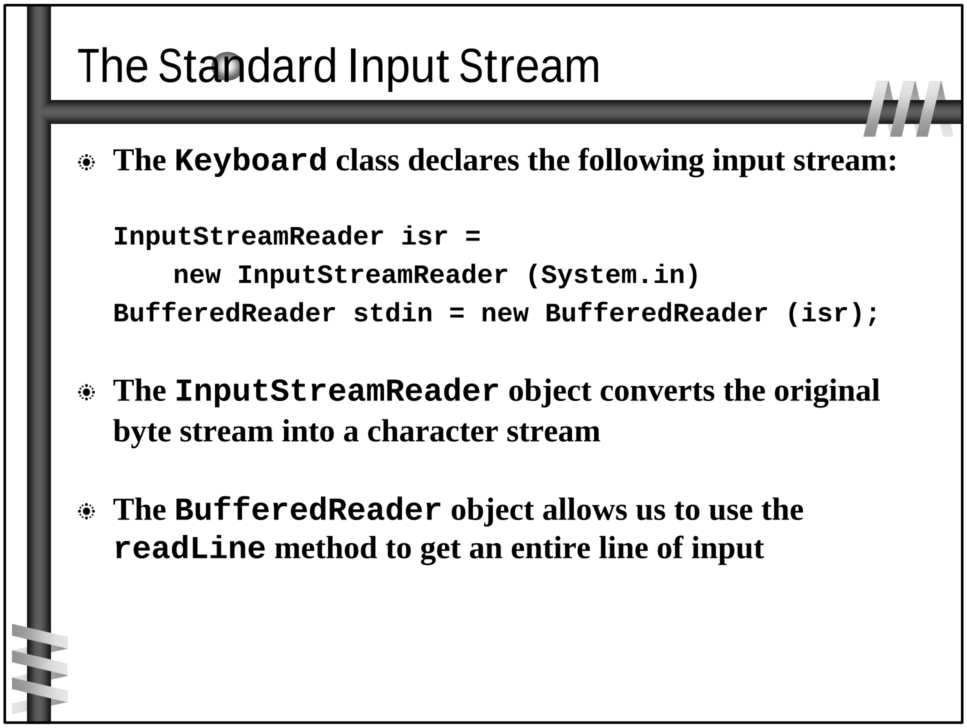
Standard I/O

- **There are three standard I/O streams:**
  - ***standard input* – defined by `System.in`**
  - ***standard output* – defined by `System.out`**
  - ***standard error* – defined by `System.err`**
- **We use `System.out` when we execute `println` statements**
- **`System.in` is declared to be a generic `InputStream` reference, and therefore usually must be mapped to a more useful stream with specific characteristics**



The Keyboard Class

- **The `Keyboard` class was written by the authors of your textbook to facilitate reading data from standard input**
- **Now we can examine the processing of the `Keyboard` class in more detail**
- **The `Keyboard` class:**
  - **declares a useful standard input stream**
  - **handles exceptions that may be thrown**
  - **parses input lines into separate values**
  - **converts input stings into the expected type**
  - **handles conversion problems**

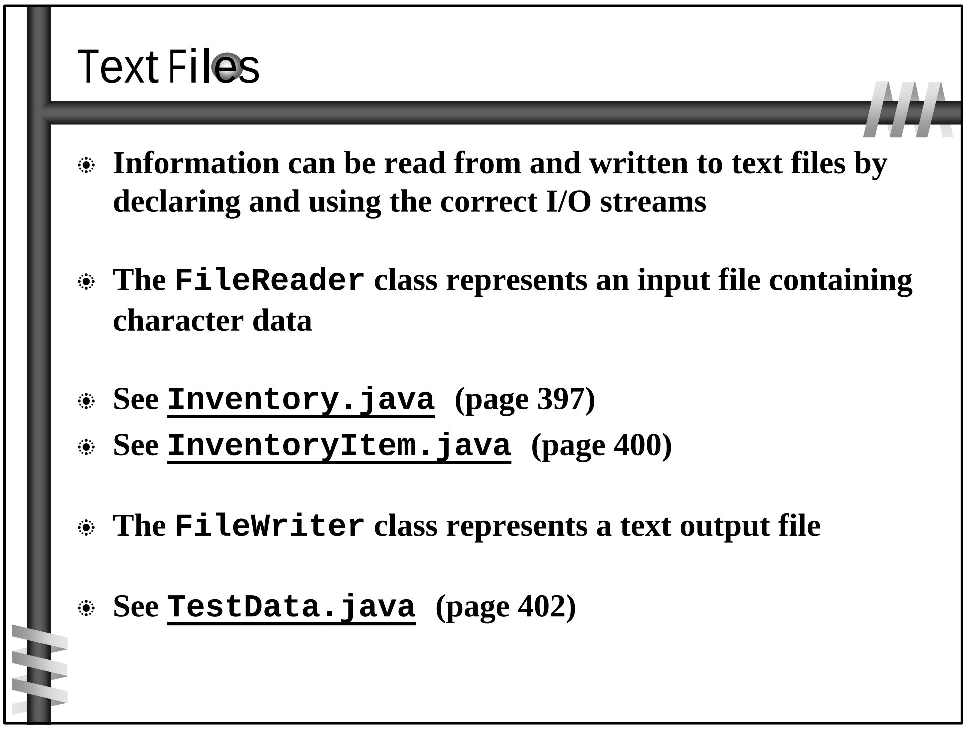


The Standard Input Stream

- The `Keyboard` class declares the following input stream:

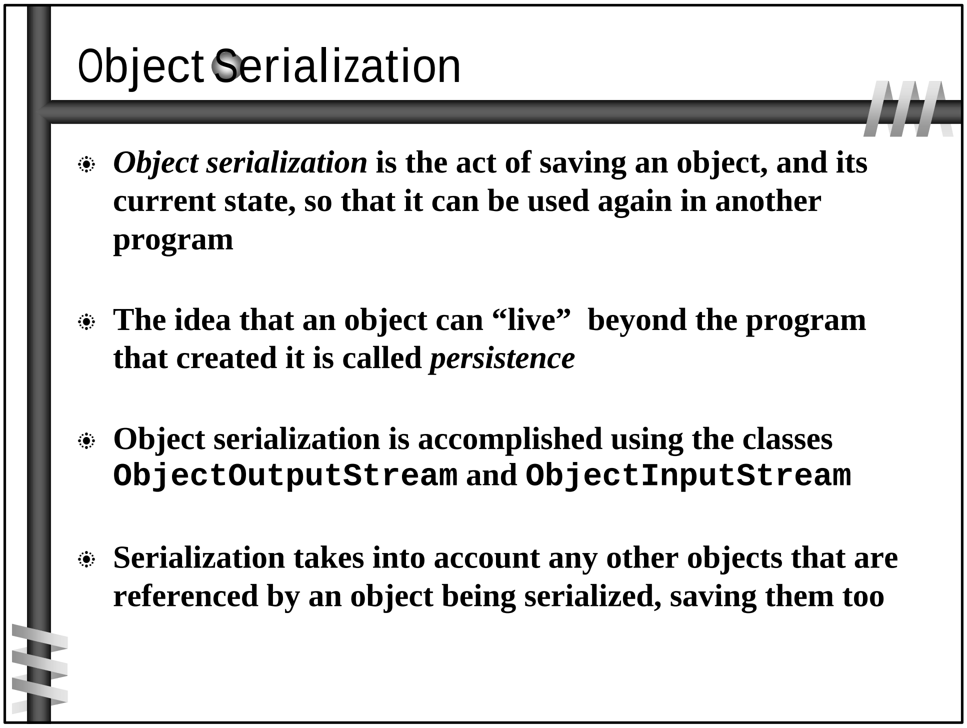
```
InputStreamReader isr =
   new InputStreamReader (System.in)
BufferedReader stdin = new BufferedReader (isr);
```

- The `InputStreamReader` object converts the original byte stream into a character stream
- The `BufferedReader` object allows us to use the `readLine` method to get an entire line of input



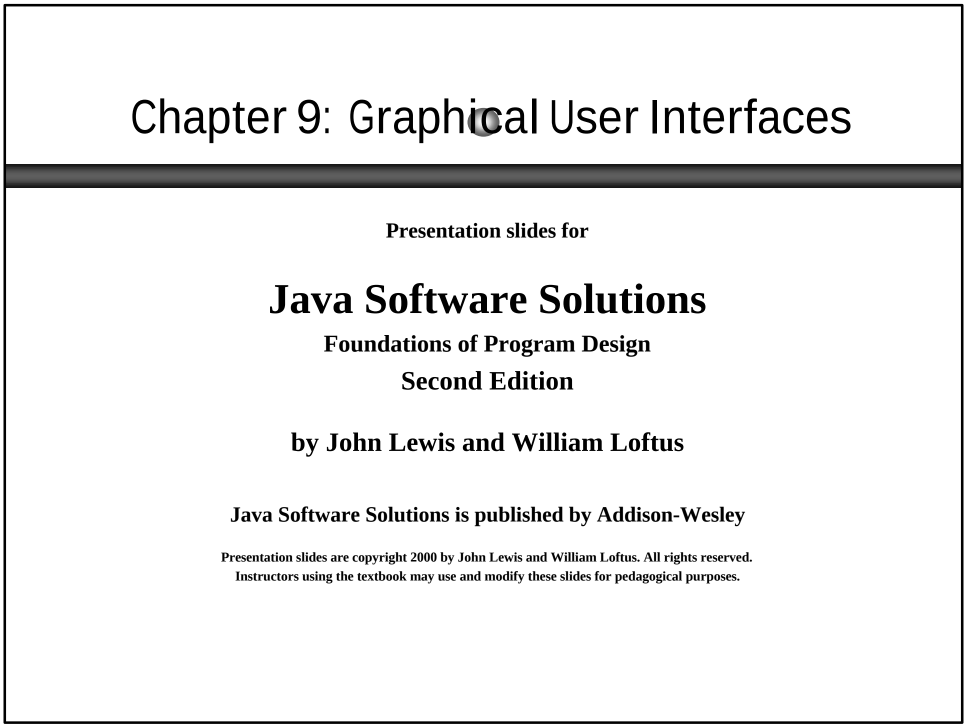
Text Files

- Information can be read from and written to text files by declaring and using the correct I/O streams
- The `FileReader` class represents an input file containing character data
- See `Inventory.java` (page 397)
- See `InventoryItem.java` (page 400)
- The `FileWriter` class represents a text output file
- See `TestData.java` (page 402)



Object Serialization

- *Object serialization* is the act of saving an object, and its current state, so that it can be used again in another program
- The idea that an object can “live” beyond the program that created it is called *persistence*
- Object serialization is accomplished using the classes `ObjectOutputStream` and `ObjectInputStream`
- Serialization takes into account any other objects that are referenced by an object being serialized, saving them too



Chapter 9: Graphical User Interfaces

Presentation slides for

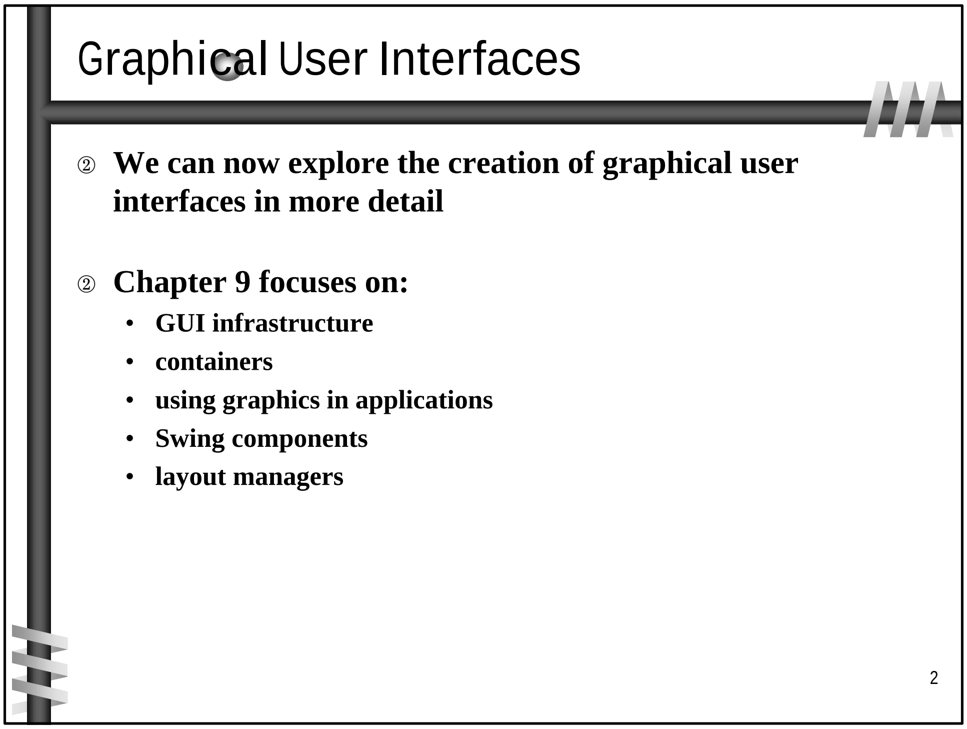
Java Software Solutions

Foundations of Program Design

Second Edition

by John Lewis and William Loftus

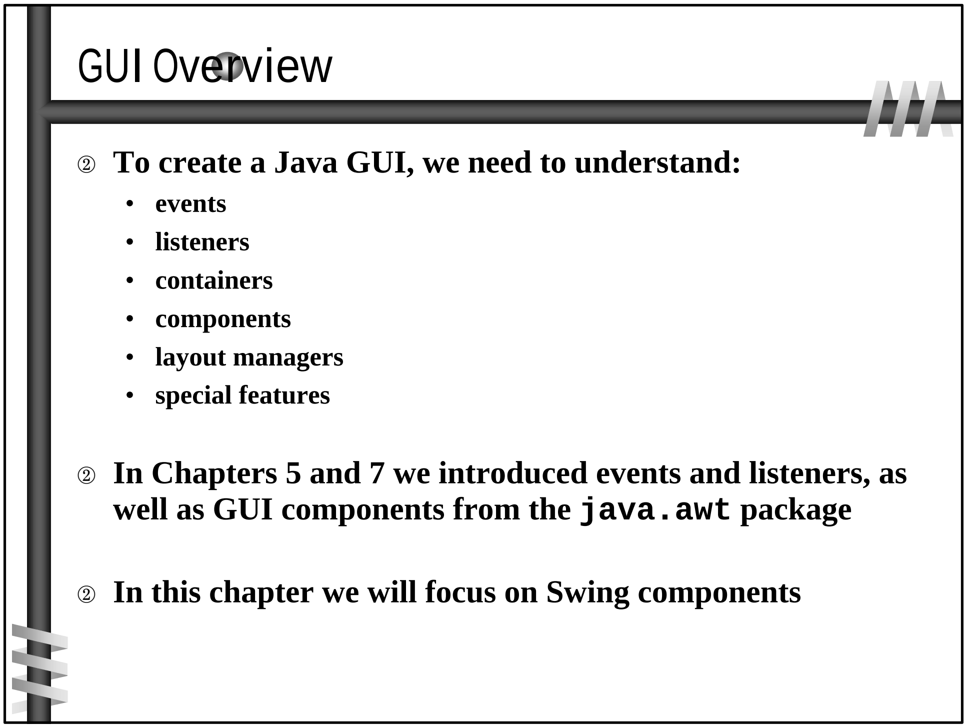
Java Software Solutions is published by Addison-Wesley



Graphical User Interfaces

- ② **We can now explore the creation of graphical user interfaces in more detail**

- ② **Chapter 9 focuses on:**
  - **GUI infrastructure**
  - **containers**
  - **using graphics in applications**
  - **Swing components**
  - **layout managers**



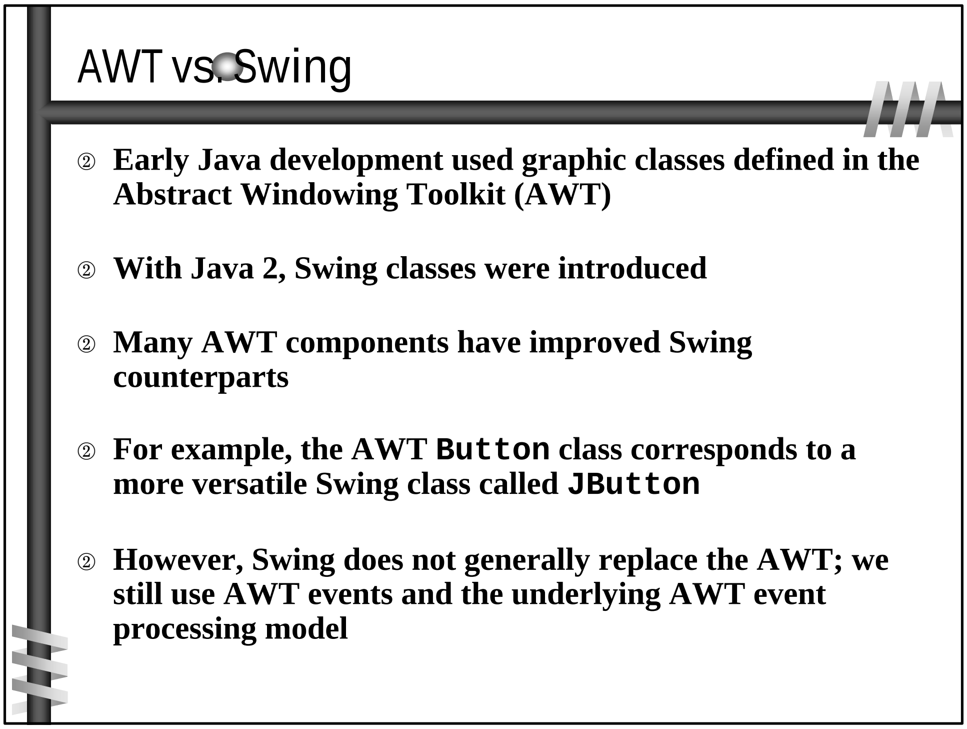
GUI Overview

② **To create a Java GUI, we need to understand:**

- **events**
- **listeners**
- **containers**
- **components**
- **layout managers**
- **special features**

② **In Chapters 5 and 7 we introduced events and listeners, as well as GUI components from the `java.awt` package**

② **In this chapter we will focus on Swing components**



AWT vs. Swing

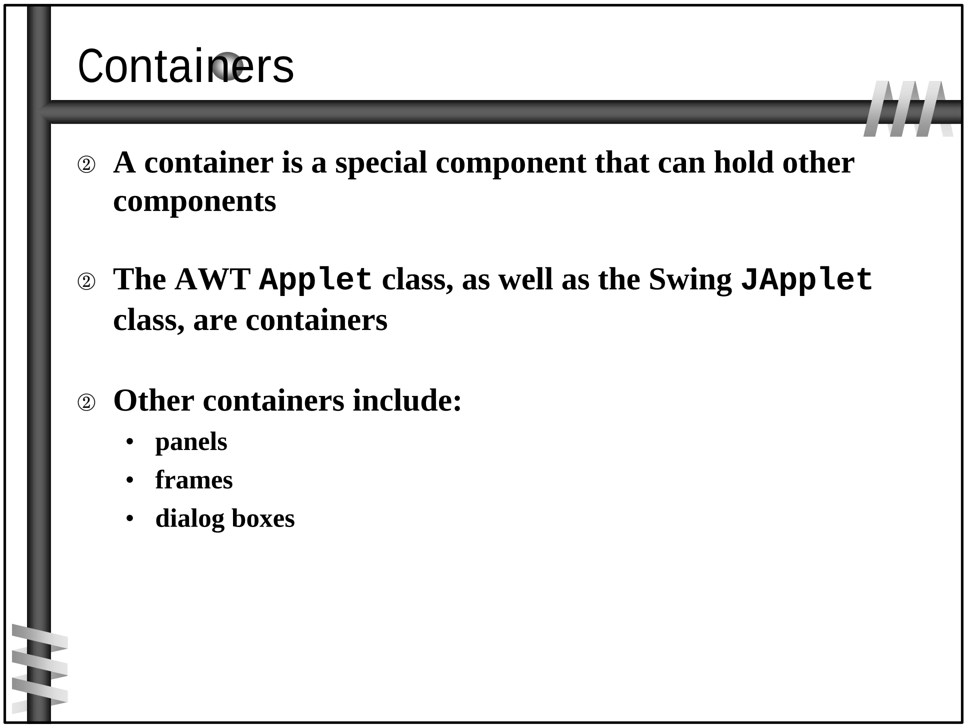
② **Early Java development used graphic classes defined in the Abstract Windowing Toolkit (AWT)**

② **With Java 2, Swing classes were introduced**

② **Many AWT components have improved Swing counterparts**

② **For example, the AWT `Button` class corresponds to a more versatile Swing class called `JButton`**

② **However, Swing does not generally replace the AWT; we still use AWT events and the underlying AWT event processing model**



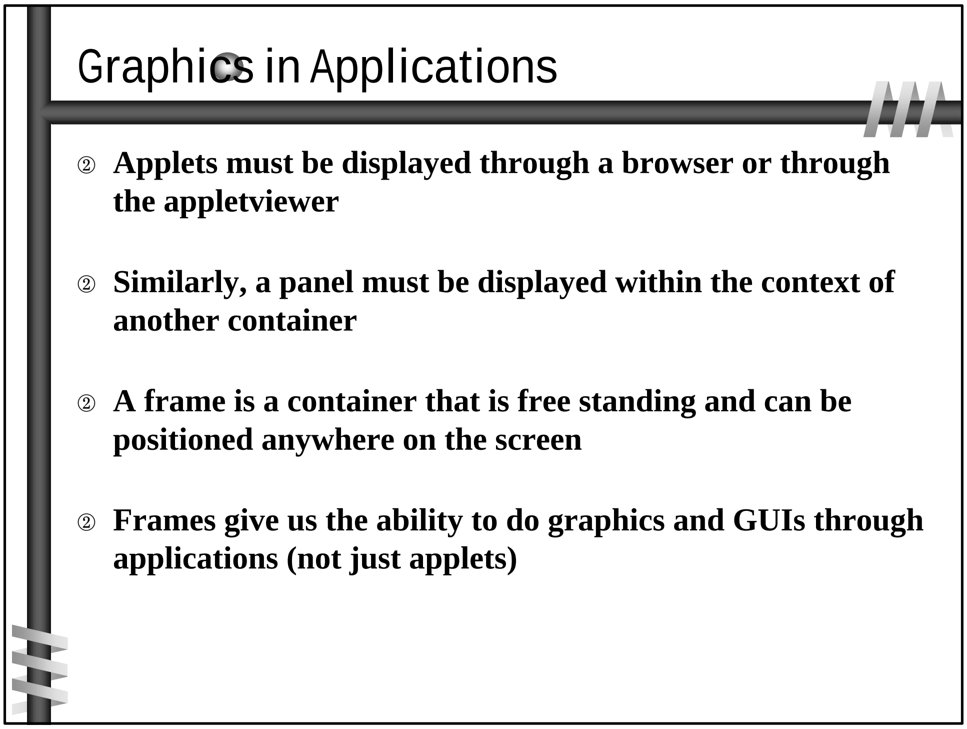
Containers

② **A container is a special component that can hold other components**

② **The AWT `Applet` class, as well as the Swing `JApplet` class, are containers**

② **Other containers include:**

- **panels**
- **frames**
- **dialog boxes**



Graphics in Applications

② **Applets must be displayed through a browser or through the appletviewer**

② **Similarly, a panel must be displayed within the context of another container**

② **A frame is a container that is free standing and can be positioned anywhere on the screen**

② **Frames give us the ability to do graphics and GUIs through applications (not just applets)**

Window Events

- **Because a frame is a free standing window, we must now address *window events***
- **Specifically, we must be able to handle a *window closing* event**
- **Frames have an icon in the corner of the window to close it**
- **Clicking it will cause the `windowClosing` method of a window listener object to be invoked**
- **See GenericWindowListener.java (page 412)**
- **See ShowFrames.java (page 413)**

Swing Components

- **There are various Swing GUI components that we can incorporate into our software:**
  - **labels (including images)**
  - **text fields and text areas**
  - **buttons**
  - **check boxes**
  - **radio buttons**
  - **menus**
  - **combo boxes**
  - **and many more…**
- **Using the proper components for the situation is an important part of GUI design**

Labels and Image Icons

② **A label is used to provide information to the user or to add decoration to the GUI**

② **A Swing label is defined by the `JLabel` class**

② **It can incorporate an image defined by the `ImageIcon` class**

② **The alignment and relative positioning of the text and image of a label can be explicitly set**

② **See ShowLabels.java (page 416)**

② **See LabelDemo.java (page 417)**

Buttons

② **GUI buttons fall into various categories:**

- **push button – a generic button that initiates some action**
- **check box – a button that can be toggled on or off**
- **radio buttons – a set of buttons that provide a set of mutually exclusive options**

② **Radio buttons must work as a group; only one can be toggled on at a time**

② **Radio buttons are grouped using the `ButtonGroup` class**

Buttons

② **Push buttons and radio buttons generate action events when pushed or toggled**

② **Check boxes generate *item state changed* events when toggled**

② **See Quotes.java (page 419)**

② **See QuotesControls.java (page 420)**

Combo Boxes

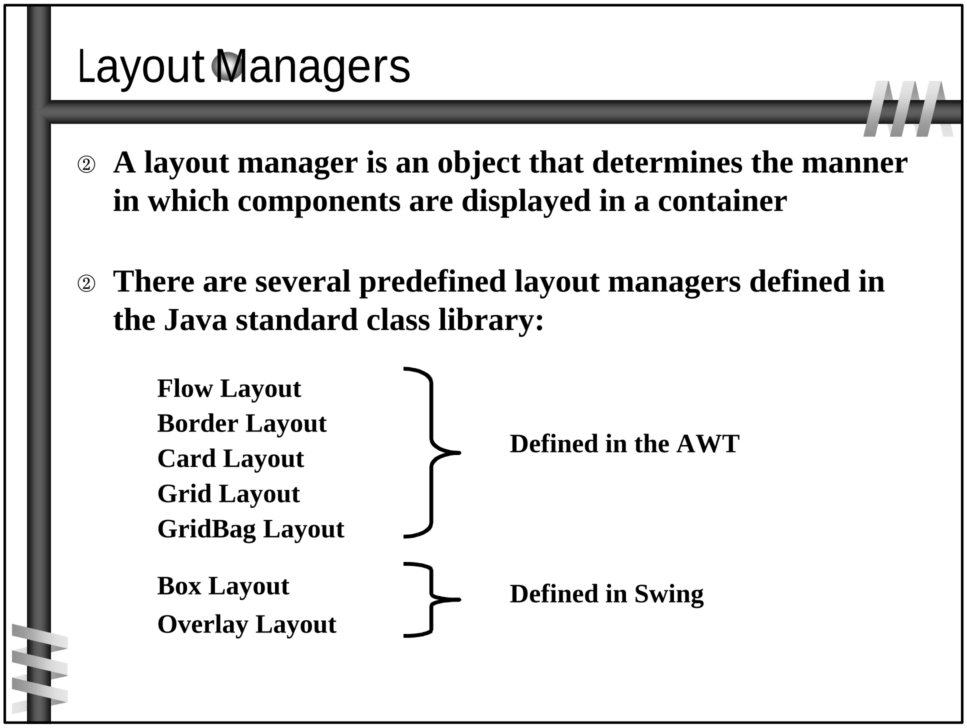
② **A *combo box* displays a particular option with a pull down menu from which the user can choose a different option**

② **The currently selected option is shown in the combo box**

② **A combo box can be *editable,* so that the user can type their option directly into the box**

② **See JukeBox.java (page 425)**

② **See JukeBoxControls.java (page 426)**



Layout Managers

② A layout manager is an object that determines the manner in which components are displayed in a container

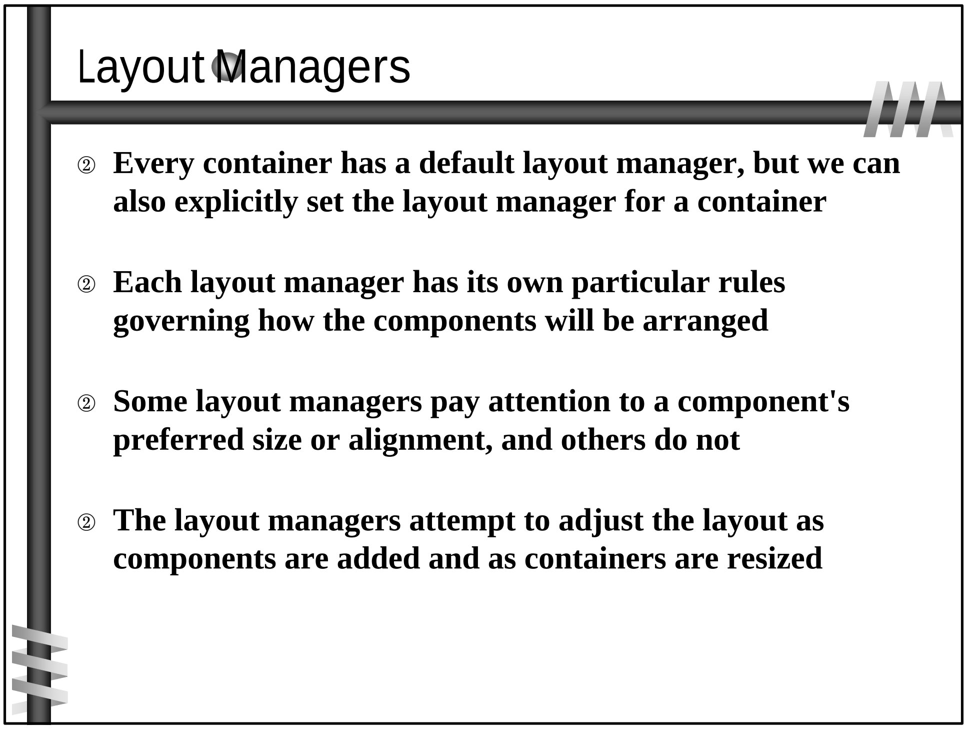
② There are several predefined layout managers defined in the Java standard class library:

Flow Layout
Border Layout
Card Layout
Grid Layout
GridBag Layout

Defined in the AWT

Box Layout
Overlay Layout

Defined in Swing



Layout Managers

② Every container has a default layout manager, but we can also explicitly set the layout manager for a container

② Each layout manager has its own particular rules governing how the components will be arranged

② Some layout managers pay attention to a component's preferred size or alignment, and others do not

② The layout managers attempt to adjust the layout as components are added and as containers are resized

Flow Layout

② A flow layout puts as many components on a row as possible, then moves to the next row

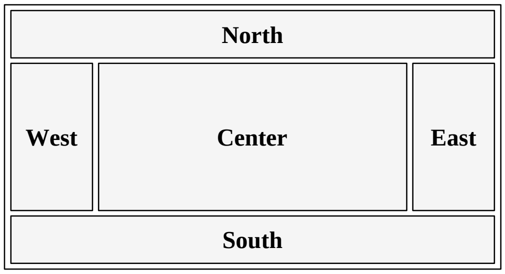
② Rows are created as needed to accommodate all of the components

② Components are displayed in the order they are added to the container

② The horizontal and vertical gaps between the components can be explicitly set

Border Layout

② A border layout defines five areas into which components can be added



Border Layout

② Each area displays one component (which could be another container)

② Each of the four outer areas enlarge as needed to accommodate the component added to them

② If nothing is added to the outer areas, they take up no space and other areas expand to fill the void

② The center area expands to fill space as needed

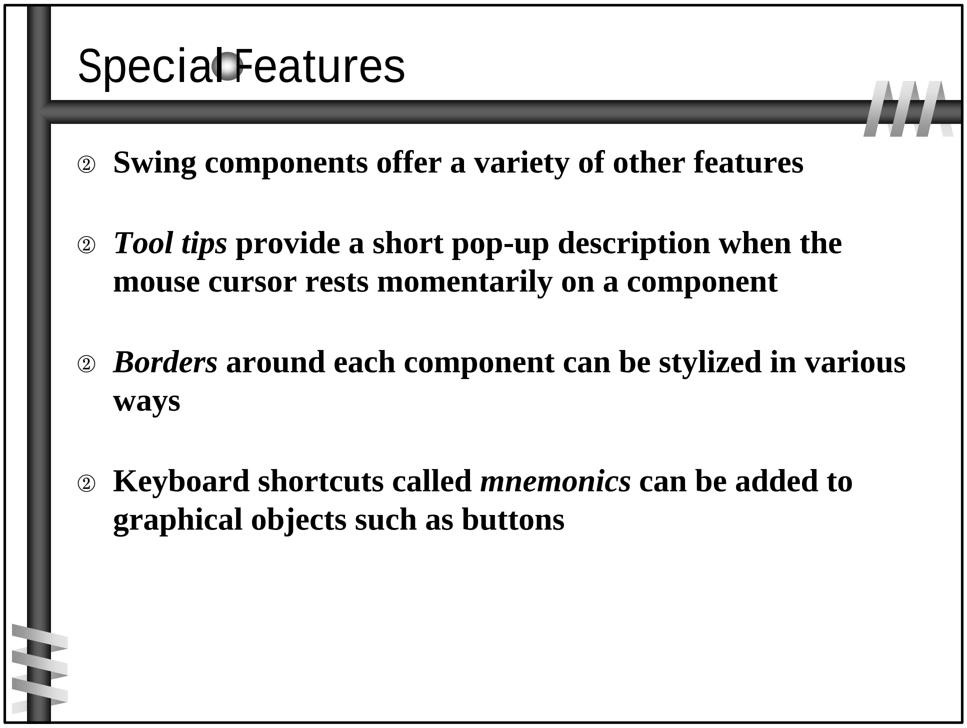
Box Layout

② A box layout organizes components either horizontally (in one row) or vertically (in one column)

② Special rigid areas can be added to force a certain amount of spacing between components

② By combining multiple containers using box layout, many different configurations can be created

② Multiple containers with box layouts are often preferred to one container that uses the more complicated gridbag layout manager



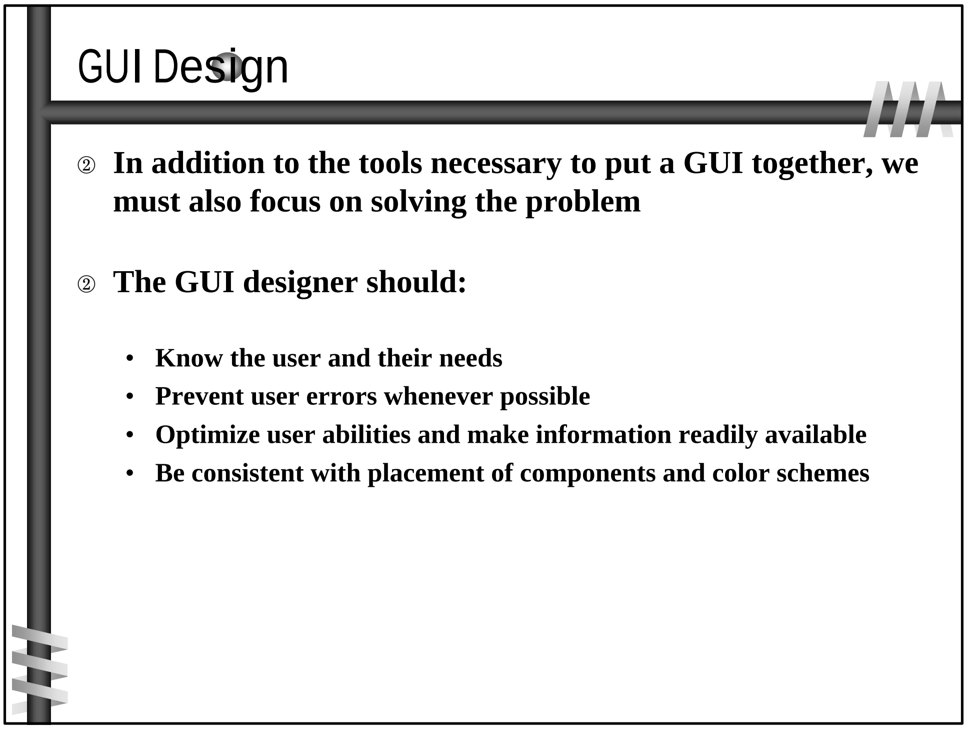
Special Features

② Swing components offer a variety of other features

② *Tool tips* provide a short pop-up description when the mouse cursor rests momentarily on a component

② *Borders* around each component can be stylized in various ways

② Keyboard shortcuts called *mnemonics* can be added to graphical objects such as buttons

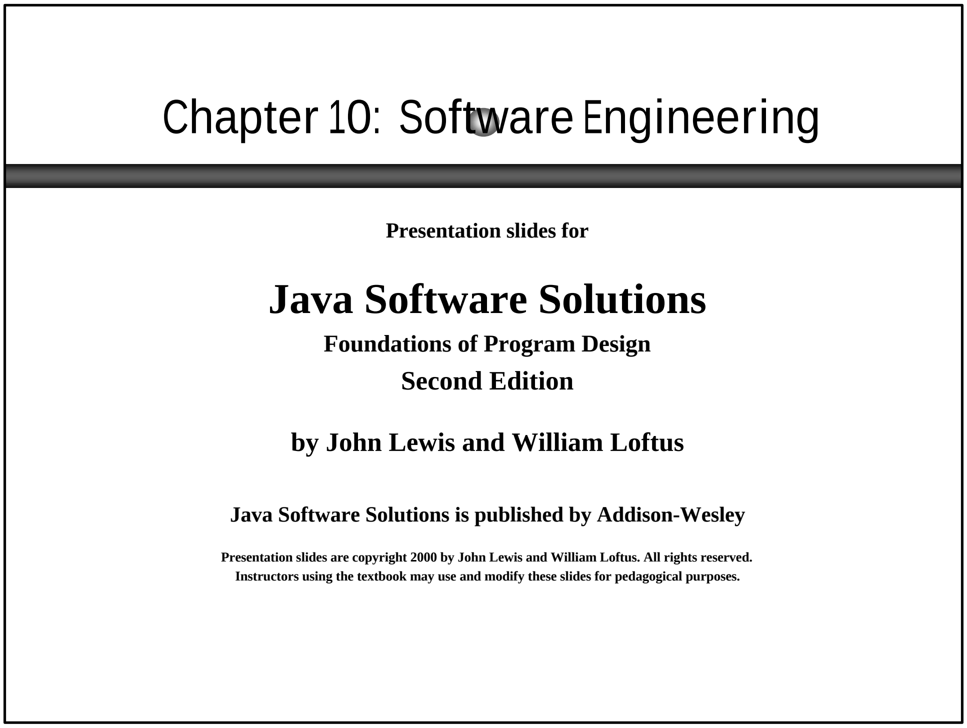


GUI Design

② In addition to the tools necessary to put a GUI together, we must also focus on solving the problem

② The GUI designer should:

- Know the user and their needs
- Prevent user errors whenever possible
- Optimize user abilities and make information readily available
- Be consistent with placement of components and color schemes



Chapter 10: Software Engineering

Presentation slides for

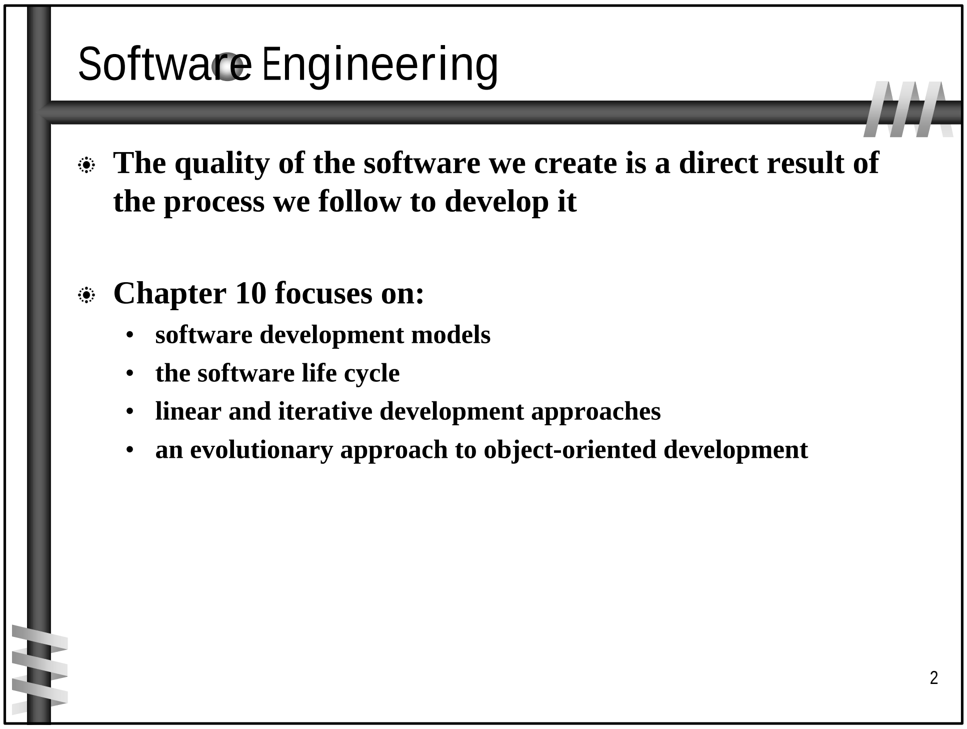
Java Software Solutions

Foundations of Program Design

Second Edition

by John Lewis and William Loftus

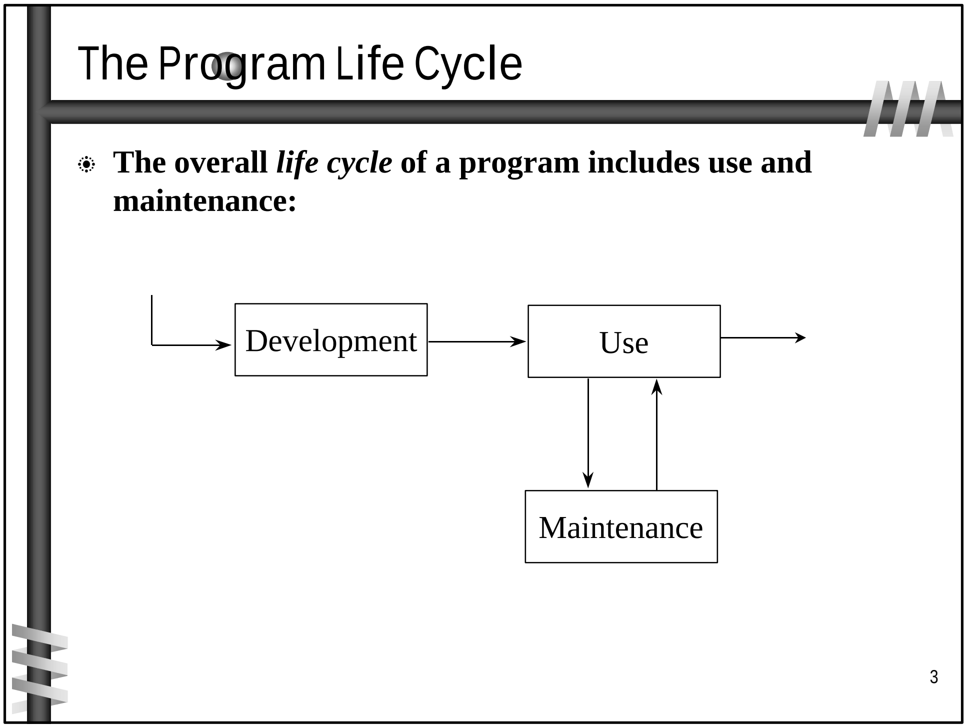
Java Software Solutions is published by Addison-Wesley



Software Engineering

- **The quality of the software we create is a direct result of the process we follow to develop it**

- **Chapter 10 focuses on:**
  - **software development models**
  - **the software life cycle**
  - **linear and iterative development approaches**
  - **an evolutionary approach to object-oriented development**



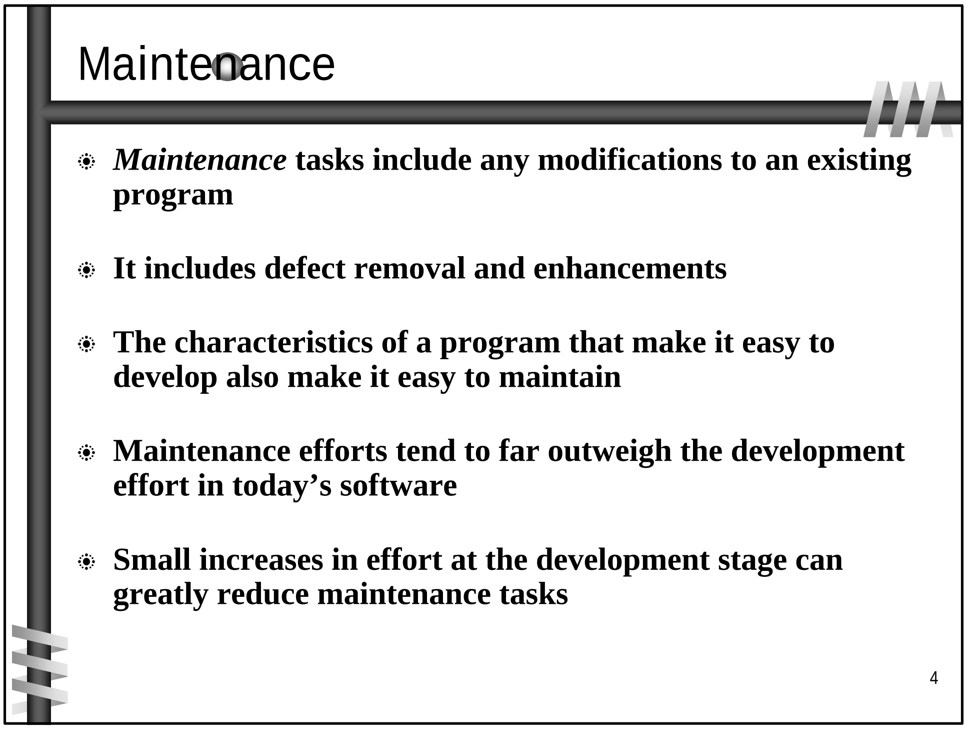
The Program Life Cycle

- The overall *life cycle* of a program includes use and maintenance:

Development

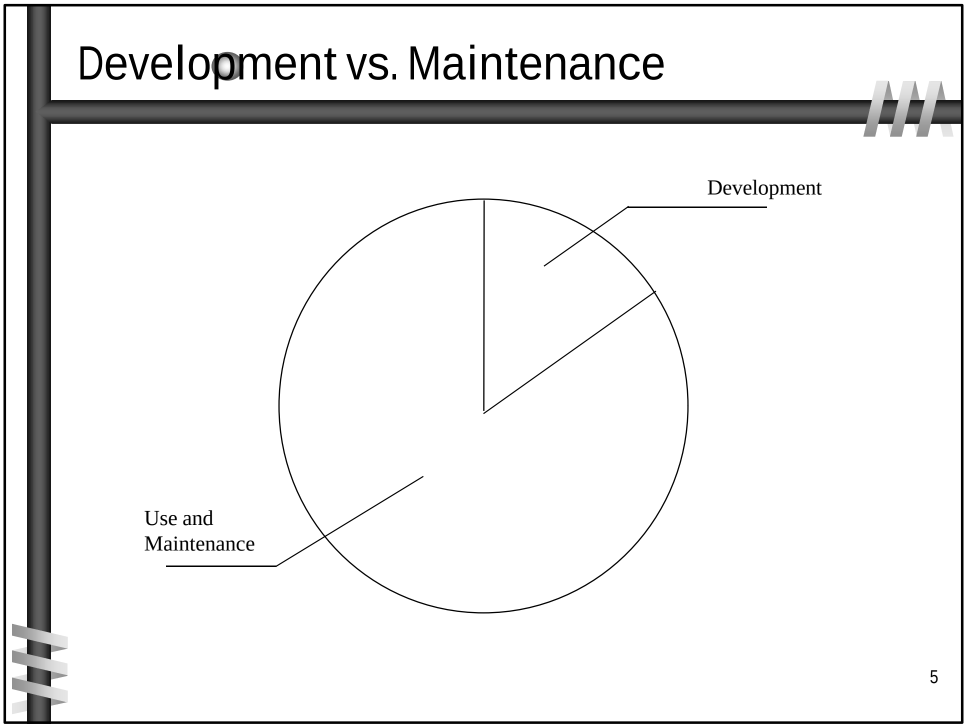
Use

Maintenance



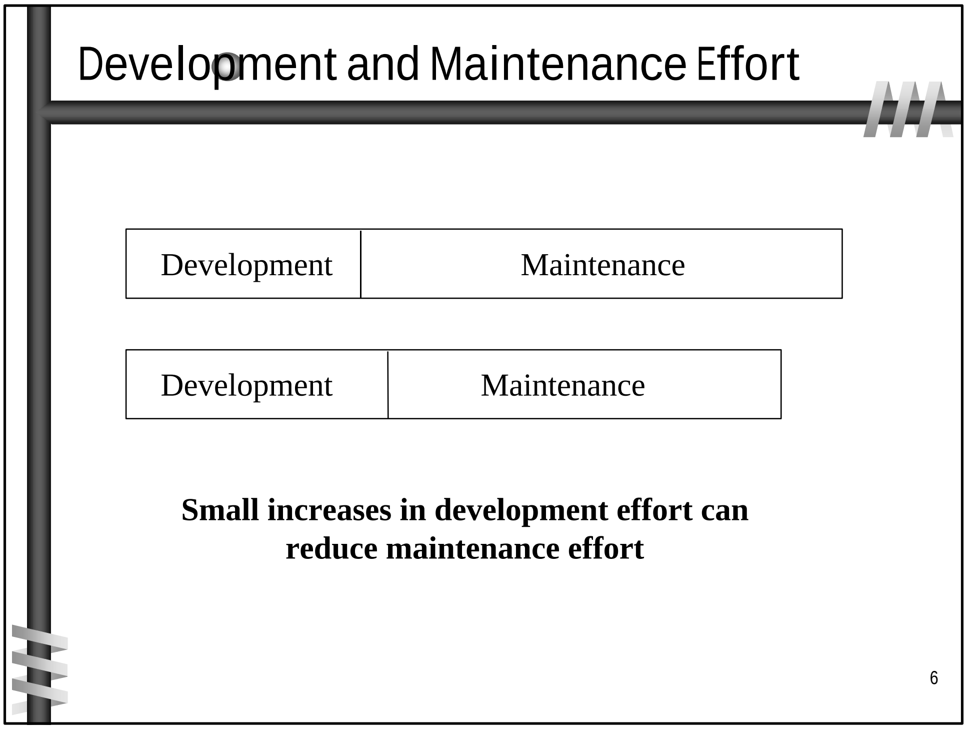
Maintenance

- *Maintenance* tasks include any modifications to an existing program
- It includes defect removal and enhancements
- The characteristics of a program that make it easy to develop also make it easy to maintain
- Maintenance efforts tend to far outweigh the development effort in today’s software
- Small increases in effort at the development stage can greatly reduce maintenance tasks



Development vs. Maintenance

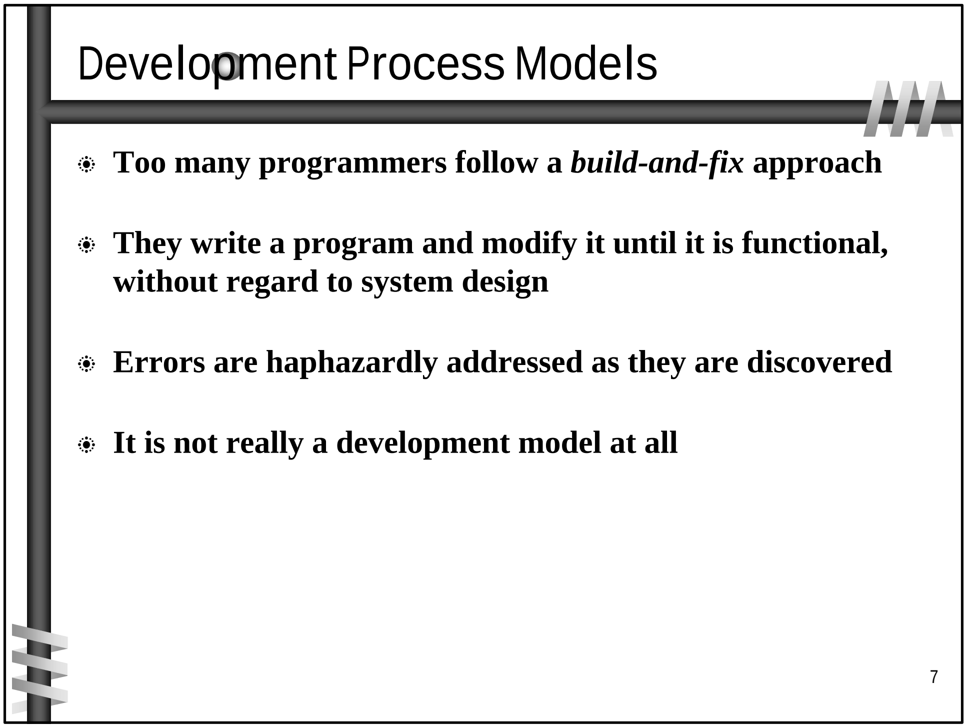
5



Development and Maintenance Effort

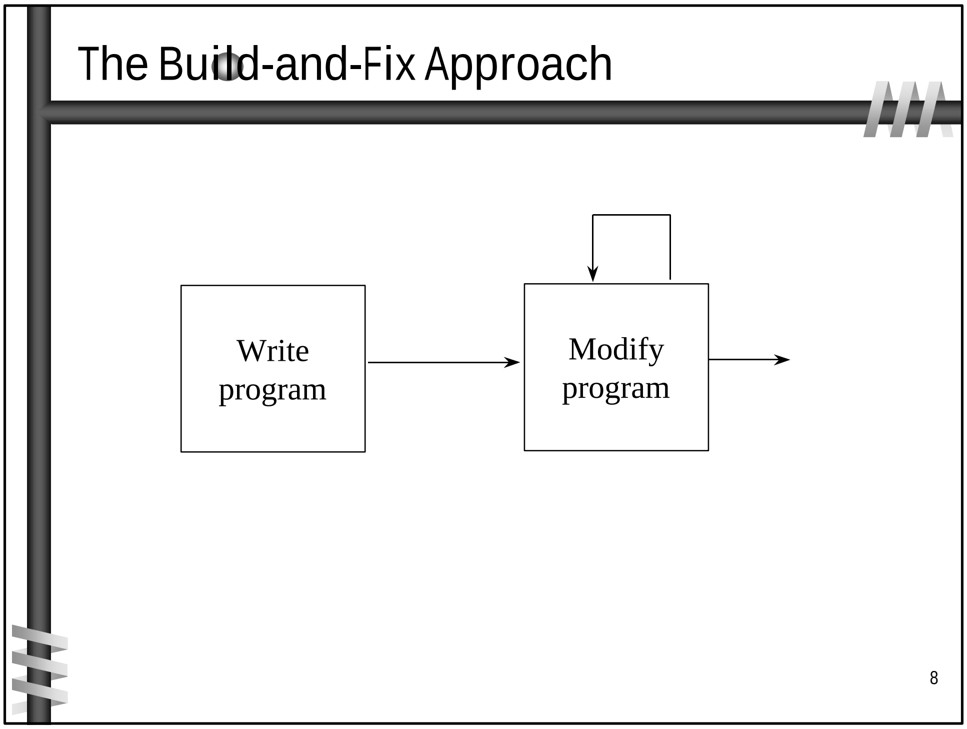
Small increases in development effort can reduce maintenance effort

6



Development Process Models

- Too many programmers follow a *build-and-fix* approach
- They write a program and modify it until it is functional, without regard to system design
- Errors are haphazardly addressed as they are discovered
- It is not really a development model at all



The Build-and-Fix Approach

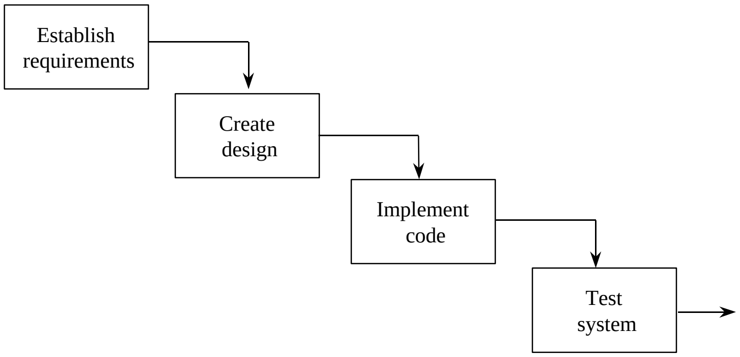
Write program

Modify program

The Waterfall Model

- The *waterfall model* was developed in the mid 1970s
- Activities that must be specifically addressed during development include:
  - Establishing clear and unambiguous requirements
  - Creating a clean design from the requirements
  - Implementing the design
  - Testing the implementation
- Originally it was proposed as a linear model, with little or no backtracking
- It is a nice goal, but is generally unrealistic

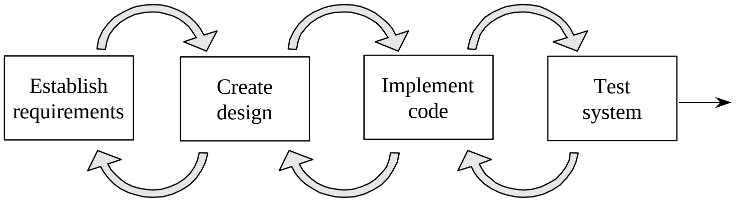
The Waterfall Model



An Iterative Process

- **Allows the developer to cycle through the different development stages**
- **Essentially the waterfall model with backtracking**
- **However backtracking should not be used irresponsibly**
- **It should be used as a technique available to the developer in order to deal with unexpected problems that may arise in later stages of development**

An Iterative Development Process

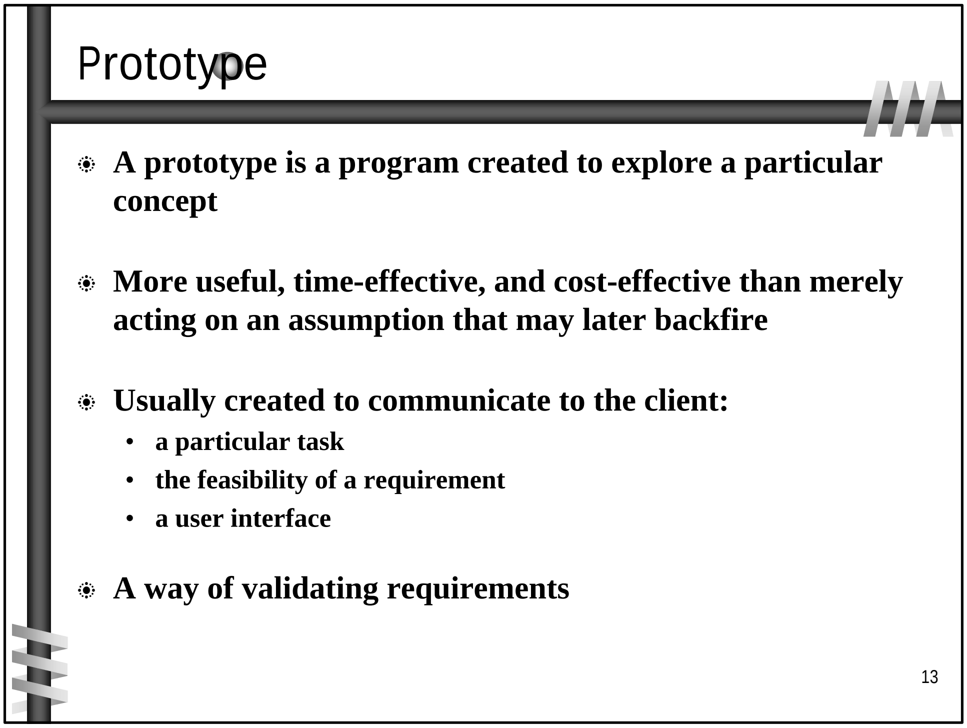


Establish requirements

Create design

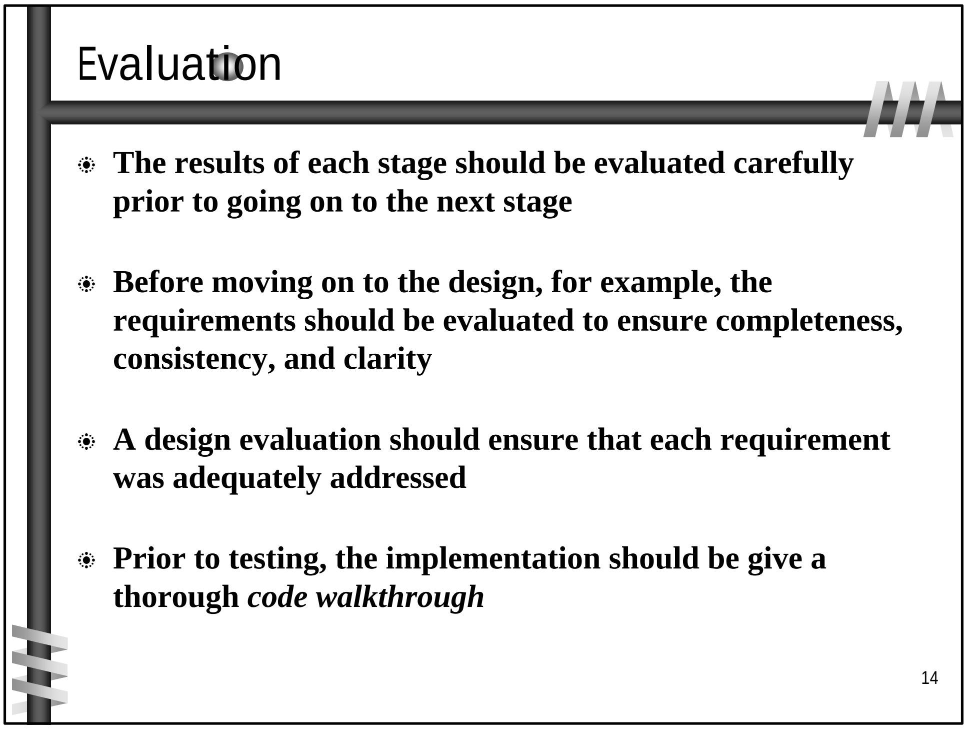
Implement code

Test system



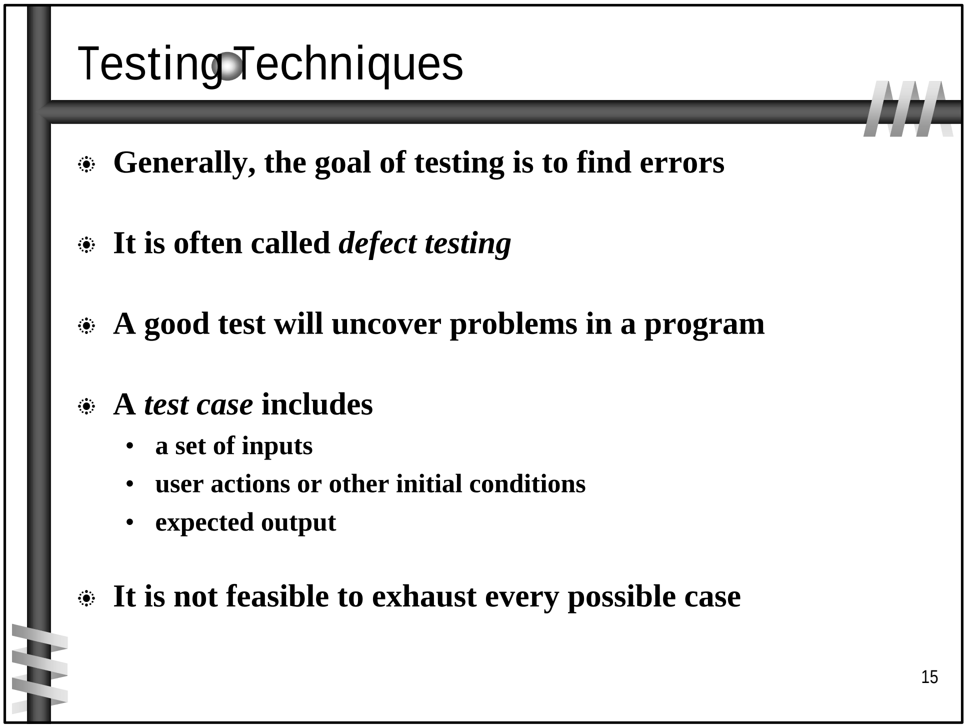
Prototype

- A prototype is a program created to explore a particular concept
- More useful, time-effective, and cost-effective than merely acting on an assumption that may later backfire
- Usually created to communicate to the client:
  - a particular task
  - the feasibility of a requirement
  - a user interface
- A way of validating requirements



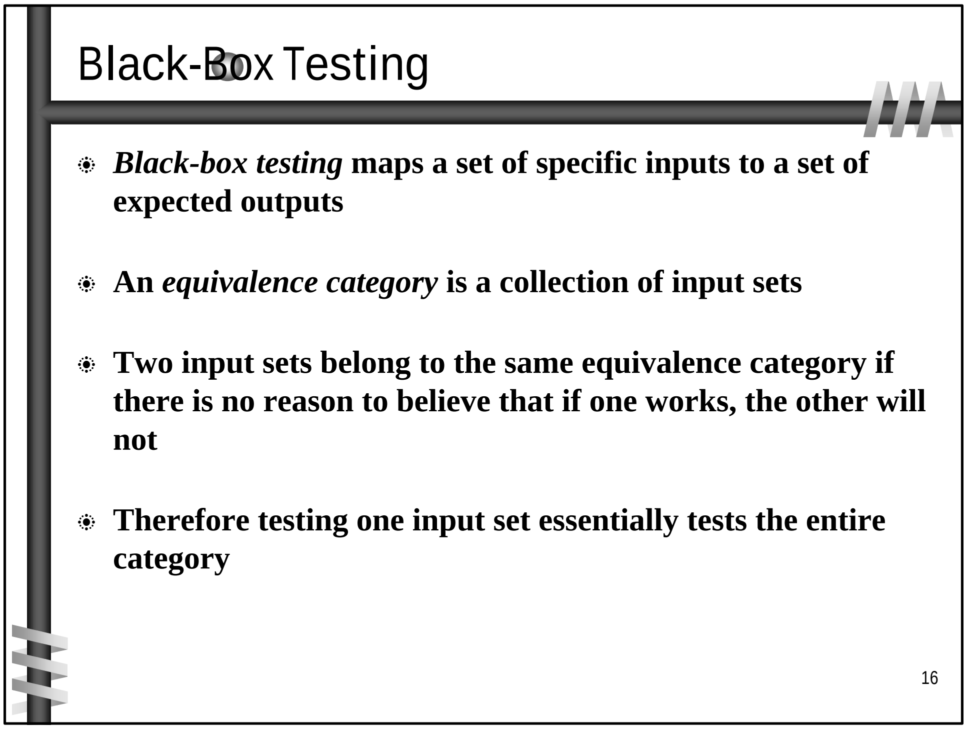
Evaluation

- The results of each stage should be evaluated carefully prior to going on to the next stage
- Before moving on to the design, for example, the requirements should be evaluated to ensure completeness, consistency, and clarity
- A design evaluation should ensure that each requirement was adequately addressed
- Prior to testing, the implementation should be give a thorough *code walkthrough*



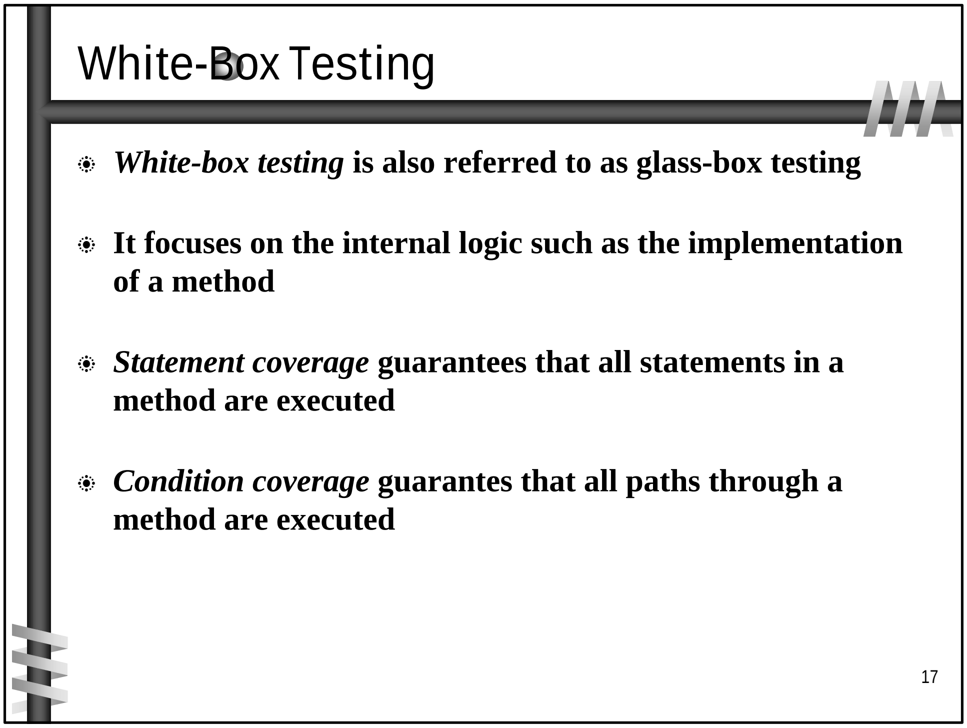
Testing Techniques

- Generally, the goal of testing is to find errors
- It is often called *defect testing*
- A good test will uncover problems in a program
- A *test case* includes
  - a set of inputs
  - user actions or other initial conditions
  - expected output
- It is not feasible to exhaust every possible case



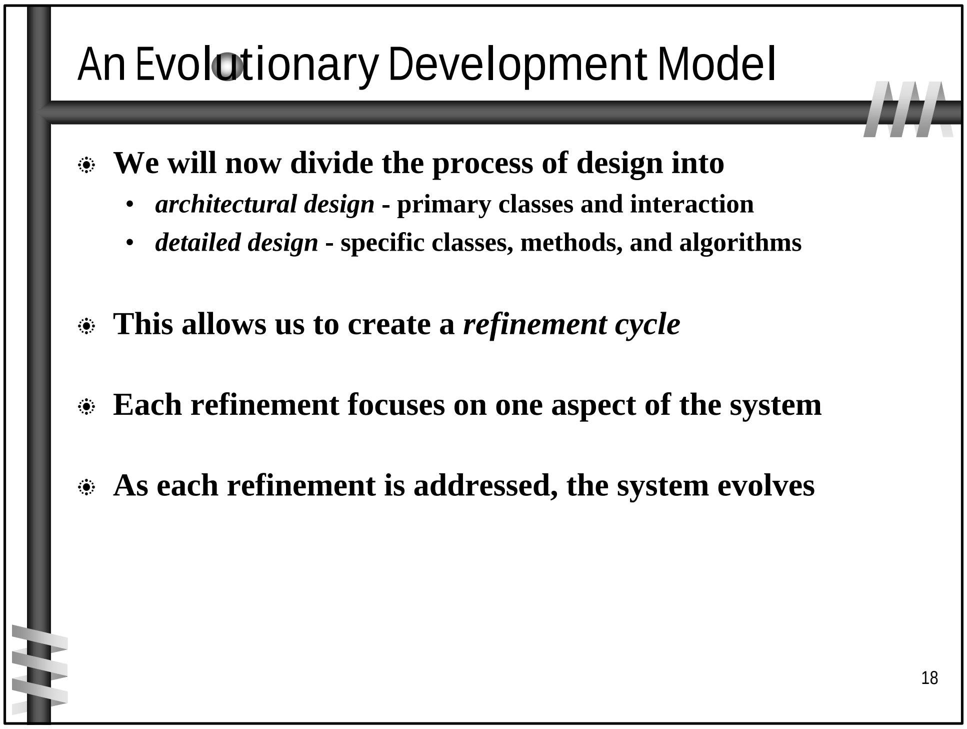
Black-Box Testing

- *Black-box testing* maps a set of specific inputs to a set of expected outputs
- An *equivalence category* is a collection of input sets
- Two input sets belong to the same equivalence category if there is no reason to believe that if one works, the other will not
- Therefore testing one input set essentially tests the entire category



White-Box Testing

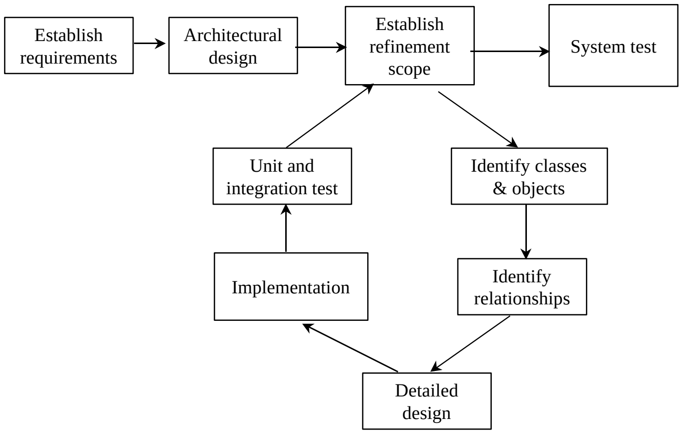
- *White-box testing* is also referred to as glass-box testing
- It focuses on the internal logic such as the implementation of a method
- *Statement coverage* guarantees that all statements in a method are executed
- *Condition coverage* guarantes that all paths through a method are executed



An Evolutionary Development Model

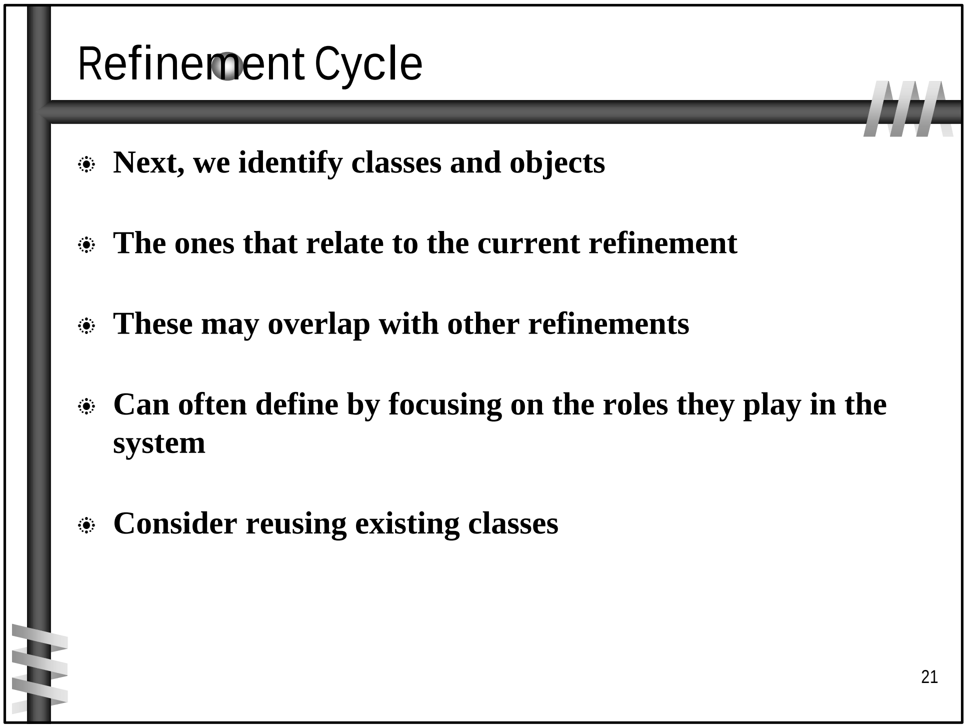
- We will now divide the process of design into
  - *architectural design* - primary classes and interaction
  - *detailed design* - specific classes, methods, and algorithms
- This allows us to create a *refinement cycle*
- Each refinement focuses on one aspect of the system
- As each refinement is addressed, the system evolves

An Evolutionary Development Model



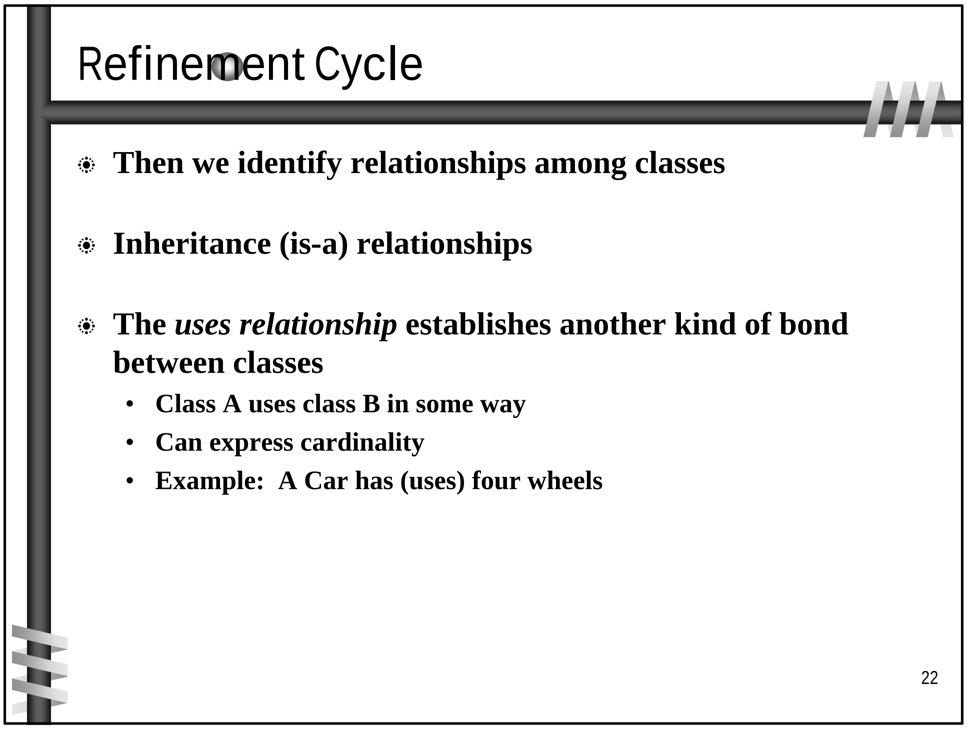
Refinement Cycle

- **First, we establish refinement scope to define the specific nature of the next refinement**
- **Such as:**
  - **the user interface**
  - **a particular algorithm**
  - **a particular requirement**
- **Choosing the most appropriate next refinement is important and requires experience**



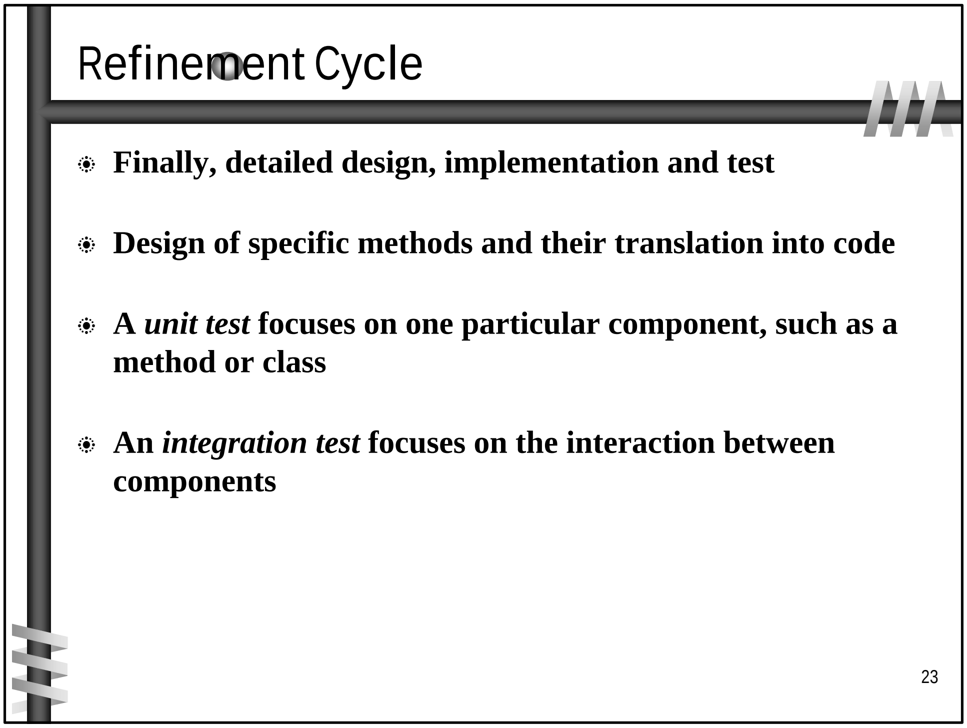
Refinement Cycle

- Next, we identify classes and objects
- The ones that relate to the current refinement
- These may overlap with other refinements
- Can often define by focusing on the roles they play in the system
- Consider reusing existing classes



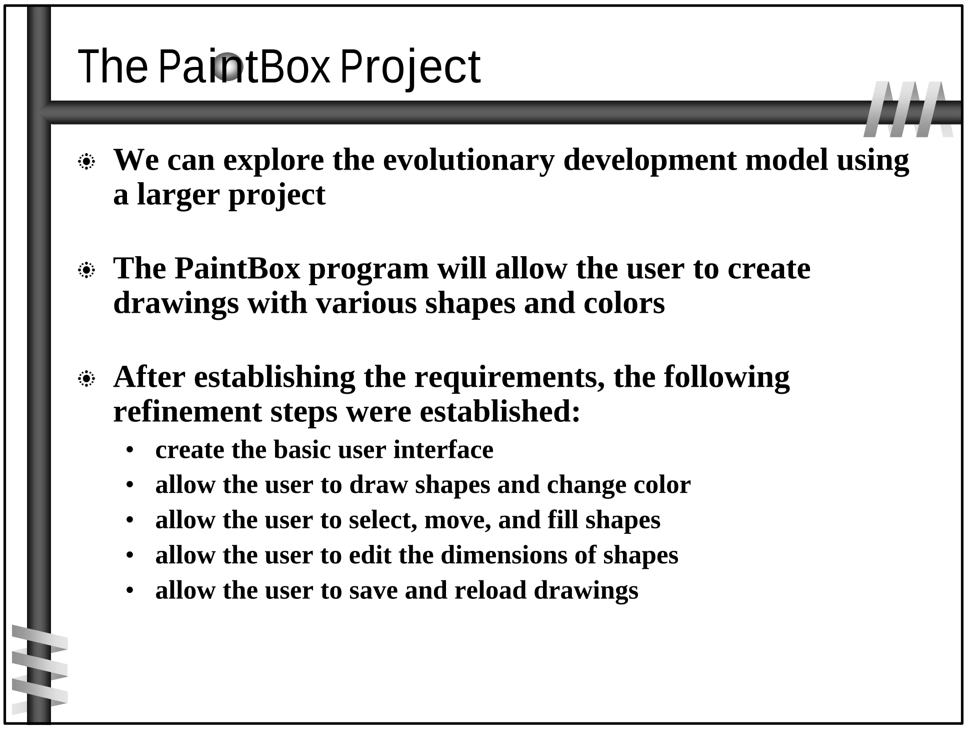
Refinement Cycle

- Then we identify relationships among classes
- Inheritance (is-a) relationships
- The *uses relationship* establishes another kind of bond between classes
  - Class A uses class B in some way
  - Can express cardinality
  - Example: A Car has (uses) four wheels



Refinement Cycle

- **Finally, detailed design, implementation and test**
- **Design of specific methods and their translation into code**
- **A *unit test* focuses on one particular component, such as a method or class**
- **An *integration test* focuses on the interaction between components**



The PaintBox Project

- **We can explore the evolutionary development model using a larger project**
- **The PaintBox program will allow the user to create drawings with various shapes and colors**
- **After establishing the requirements, the following refinement steps were established:**
  - **create the basic user interface**
  - **allow the user to draw shapes and change color**
  - **allow the user to select, move, and fill shapes**
  - **allow the user to edit the dimensions of shapes**
  - **allow the user to save and reload drawings**

PaintBox Refinement 1

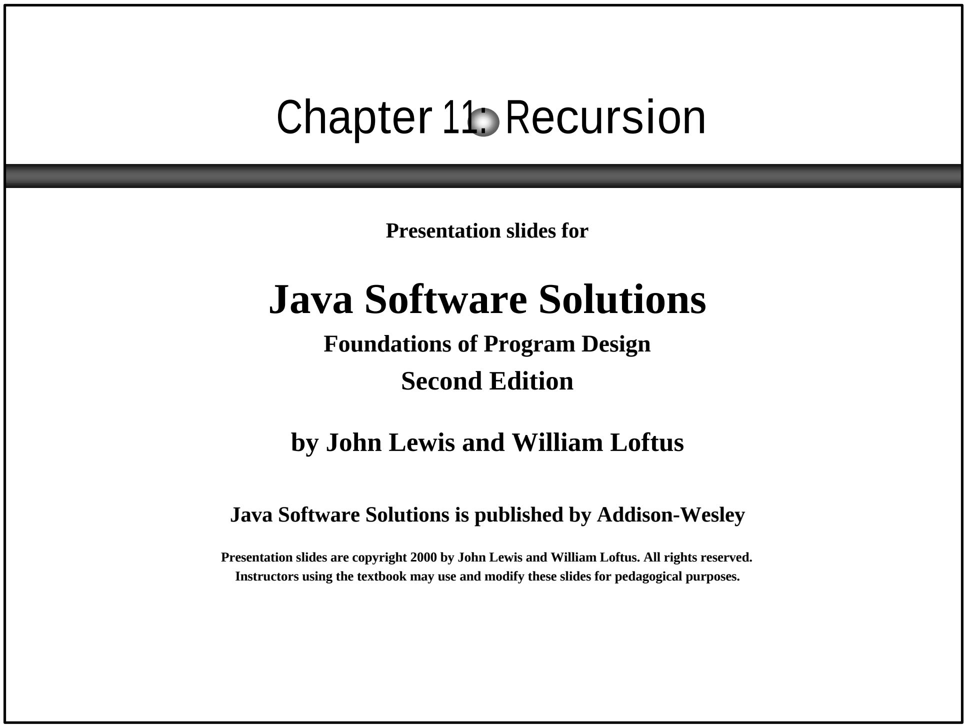
- **The first refinement establishes the basic user interface**
- **See `PaintBox.java`**
- **See `PaintFrame.java`**
- **See `ButtonPanel.java`**
- **See DrawingPanel.java**

PaintBox Refinement 2

- **The second refinement allows the user to draw shapes and change colors**

- **See `PaintBox.java`**
- **See `PaintFrame.java`**
- **See `ButtonPanel.java`**
- **See DrawingPanel.java**

- **See `Shape.java`**
- **See `Line.java`**
- **See `BoundedShape.java`**
- **See `Rect.java`**
- **See Oval.java**
- **See Poly.java**



Chapter 11: Recursion

Presentation slides for

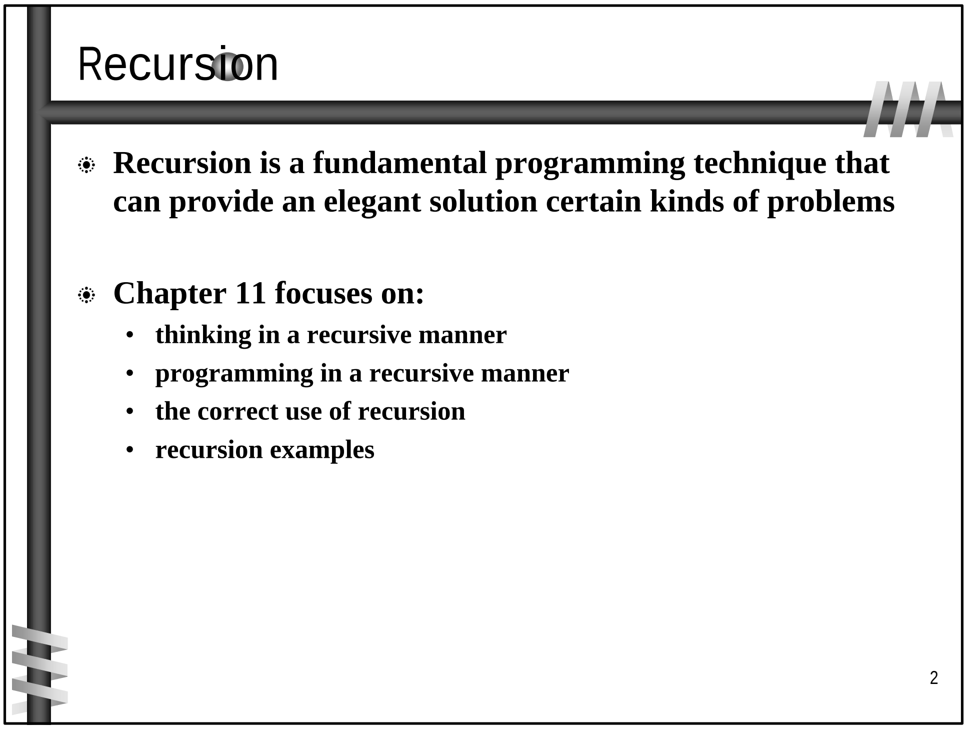
Java Software Solutions

Foundations of Program Design

Second Edition

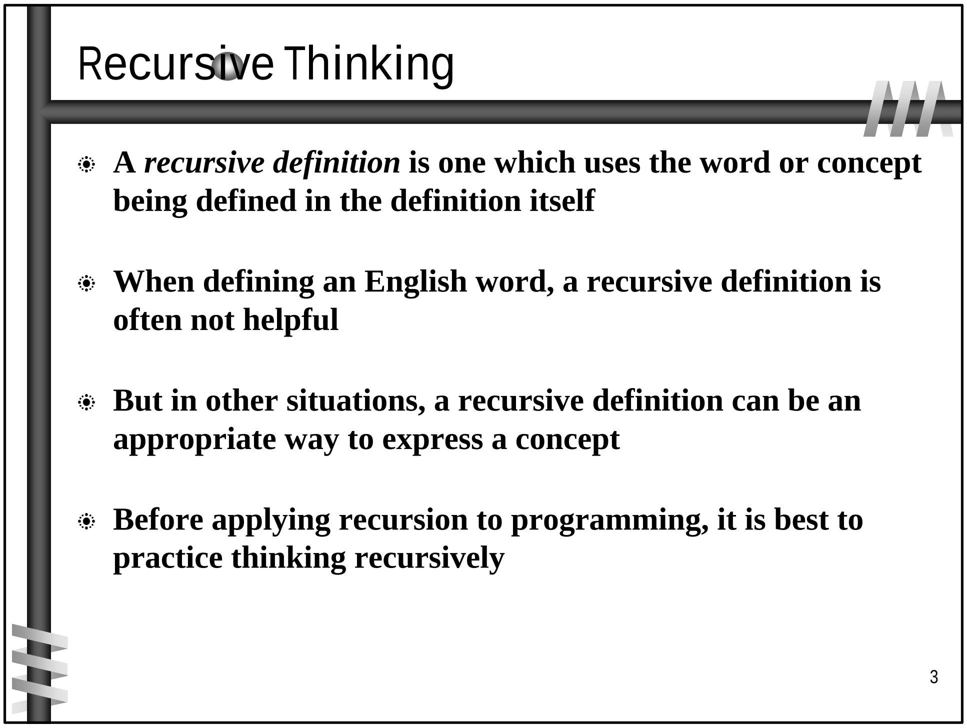
by John Lewis and William Loftus

Java Software Solutions is published by Addison-Wesley



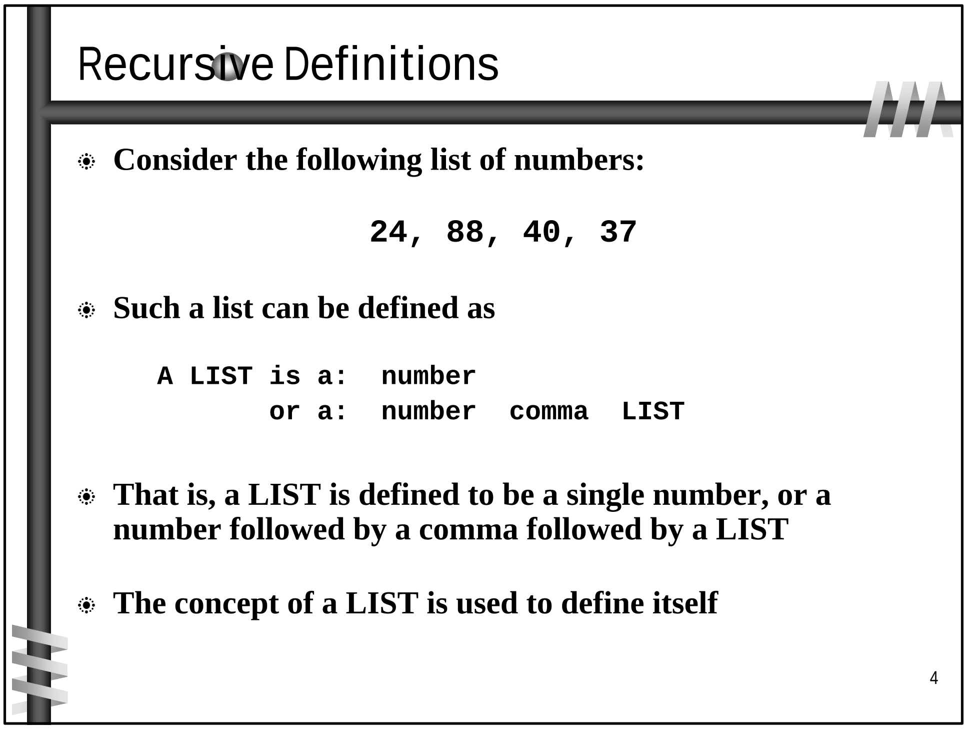
Recursion

- **Recursion is a fundamental programming technique that can provide an elegant solution certain kinds of problems**
- **Chapter 11 focuses on:**
  - **thinking in a recursive manner**
  - **programming in a recursive manner**
  - **the correct use of recursion**
  - **recursion examples**



Recursive Thinking

- A *recursive definition* is one which uses the word or concept being defined in the definition itself
- When defining an English word, a recursive definition is often not helpful
- But in other situations, a recursive definition can be an appropriate way to express a concept
- Before applying recursion to programming, it is best to practice thinking recursively



Recursive Definitions

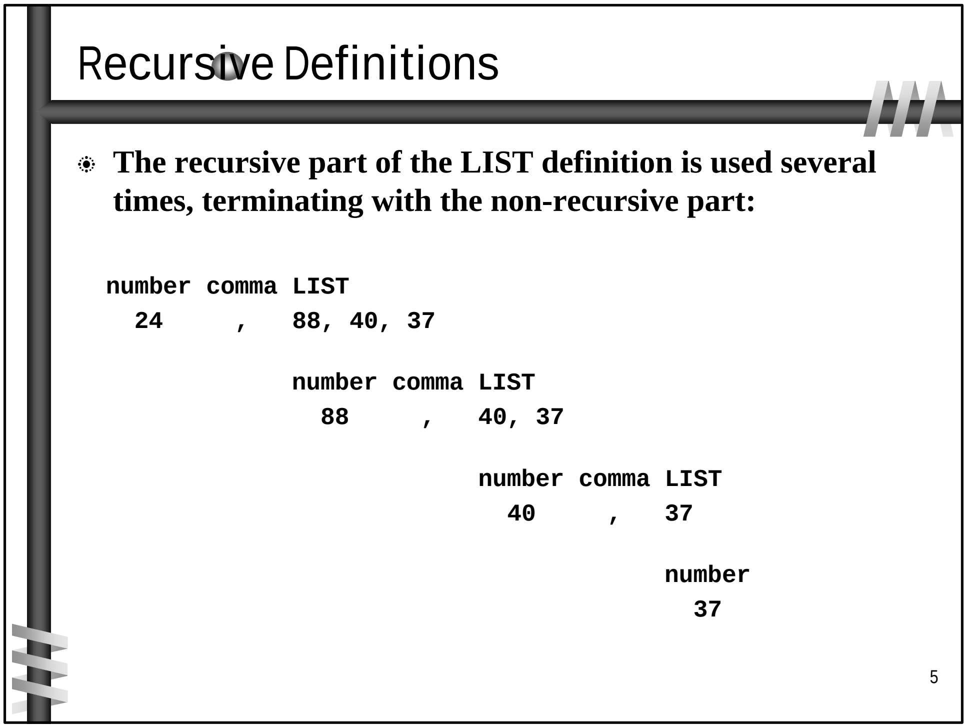
- Consider the following list of numbers:

```
24, 88, 40, 37
```

- Such a list can be defined as

```
A LIST is a:  number
       or a:  number  comma  LIST
```

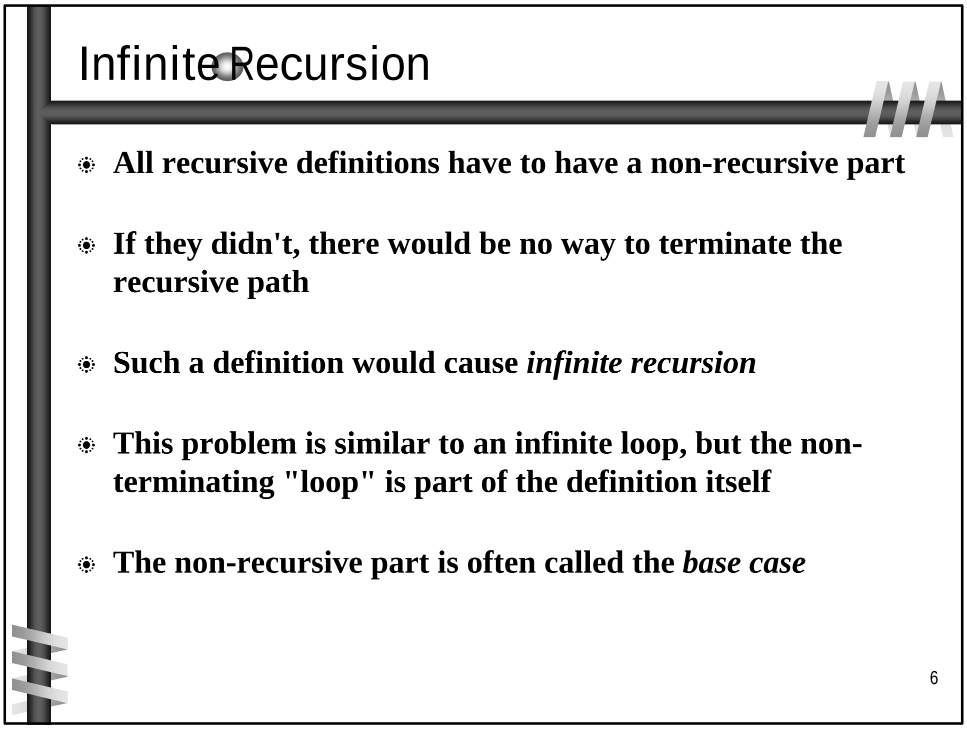
- That is, a LIST is defined to be a single number, or a number followed by a comma followed by a LIST
- The concept of a LIST is used to define itself



Recursive Definitions

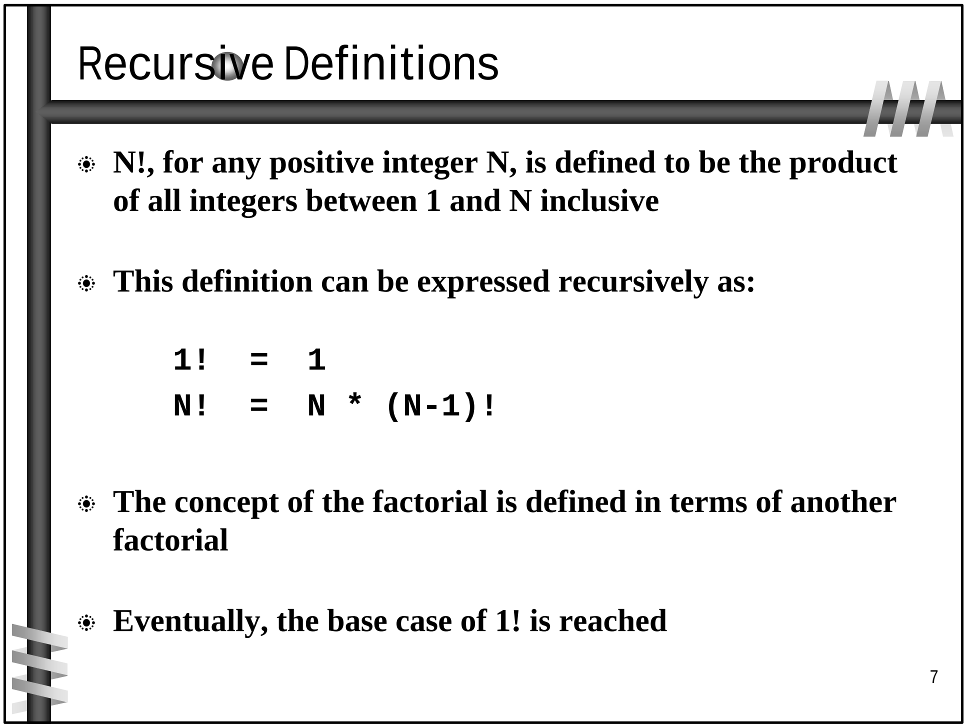
- The recursive part of the LIST definition is used several times, terminating with the non-recursive part:

```
number comma LIST
  24     ,   88, 40, 37

             number comma LIST
               88     ,   40, 37

                          number comma LIST
                            40     ,   37

                                       number
                                         37
```



Infinite Recursion

- All recursive definitions have to have a non-recursive part
- If they didn't, there would be no way to terminate the recursive path
- Such a definition would cause *infinite recursion*
- This problem is similar to an infinite loop, but the non-terminating "loop" is part of the definition itself
- The non-recursive part is often called the *base case*

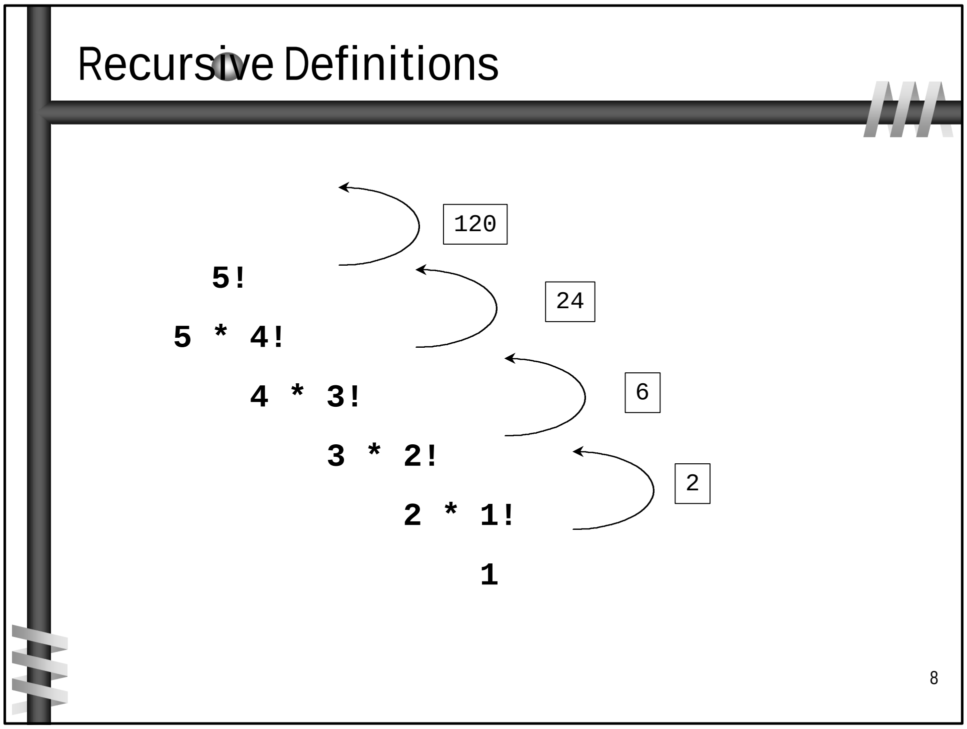


Recursive Definitions

- **N!, for any positive integer N, is defined to be the product of all integers between 1 and N inclusive**
- **This definition can be expressed recursively as:**

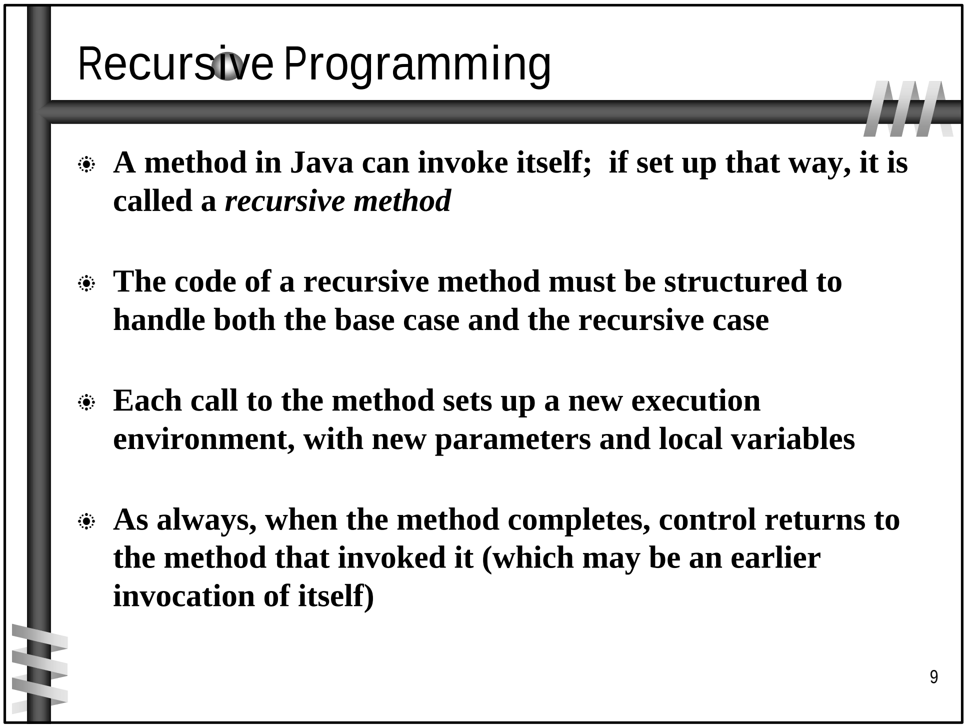
```
1!  =  1
N!  =  N * (N-1)!
```

- **The concept of the factorial is defined in terms of another factorial**
- **Eventually, the base case of 1! is reached**



Recursive Definitions

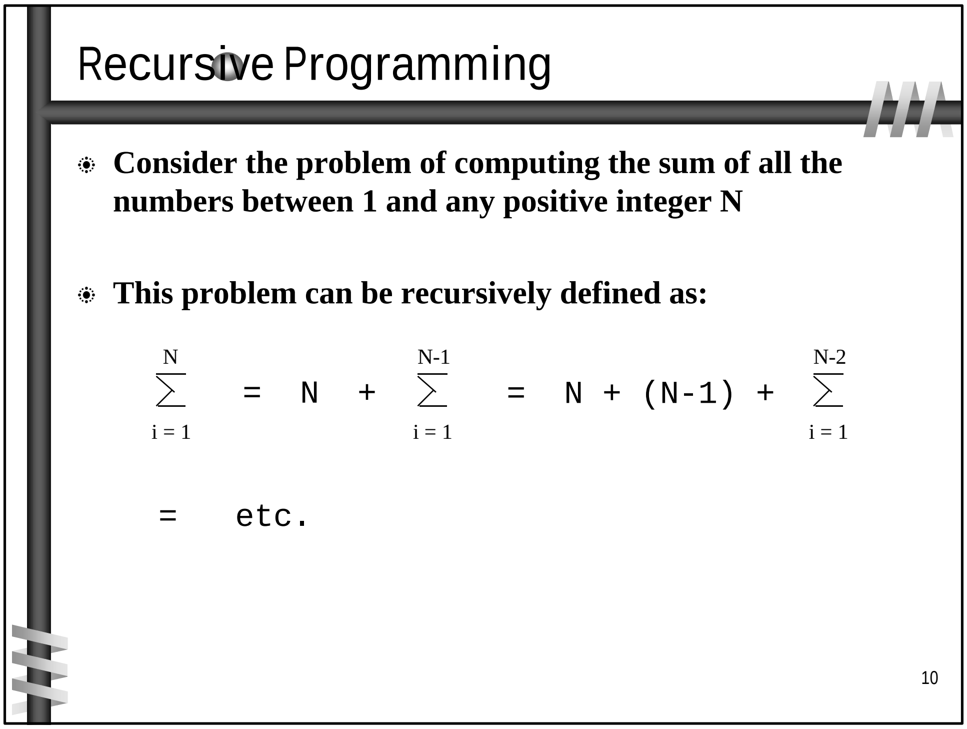
```
  5!
5 * 4!
    4 * 3!
        3 * 2!
            2 * 1!
                1
```



Recursive Programming

- A method in Java can invoke itself; if set up that way, it is called a *recursive method*
- The code of a recursive method must be structured to handle both the base case and the recursive case
- Each call to the method sets up a new execution environment, with new parameters and local variables
- As always, when the method completes, control returns to the method that invoked it (which may be an earlier invocation of itself)

9



Recursive Programming

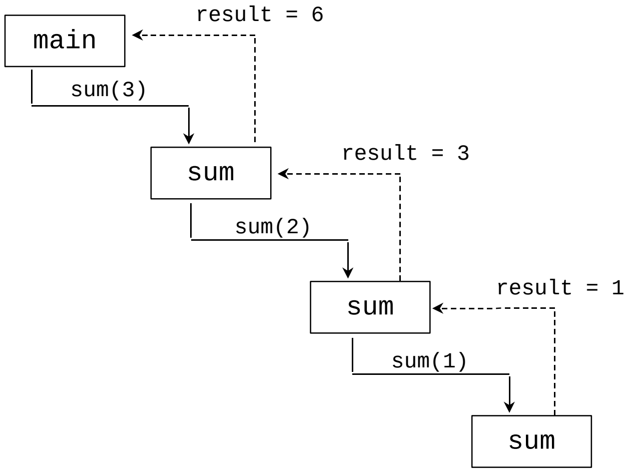
- Consider the problem of computing the sum of all the numbers between 1 and any positive integer N
- This problem can be recursively defined as:

$$\sum_{i=1}^{N} = N + \sum_{i=1}^{N-1} = N + (N-1) + \sum_{i=1}^{N-2}$$

$$= \text{etc.}$$

10

Recursive Programming



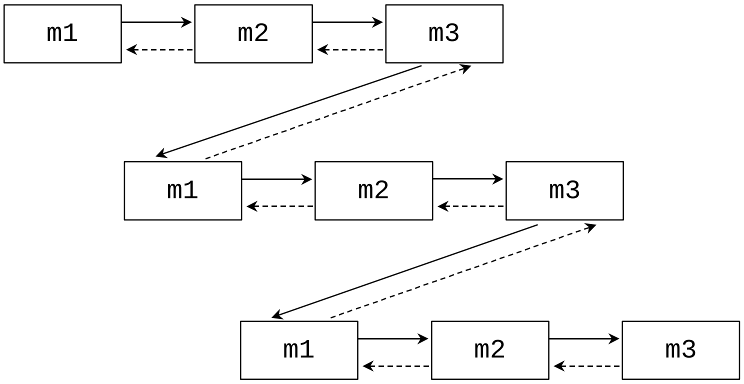
Recursive Programming

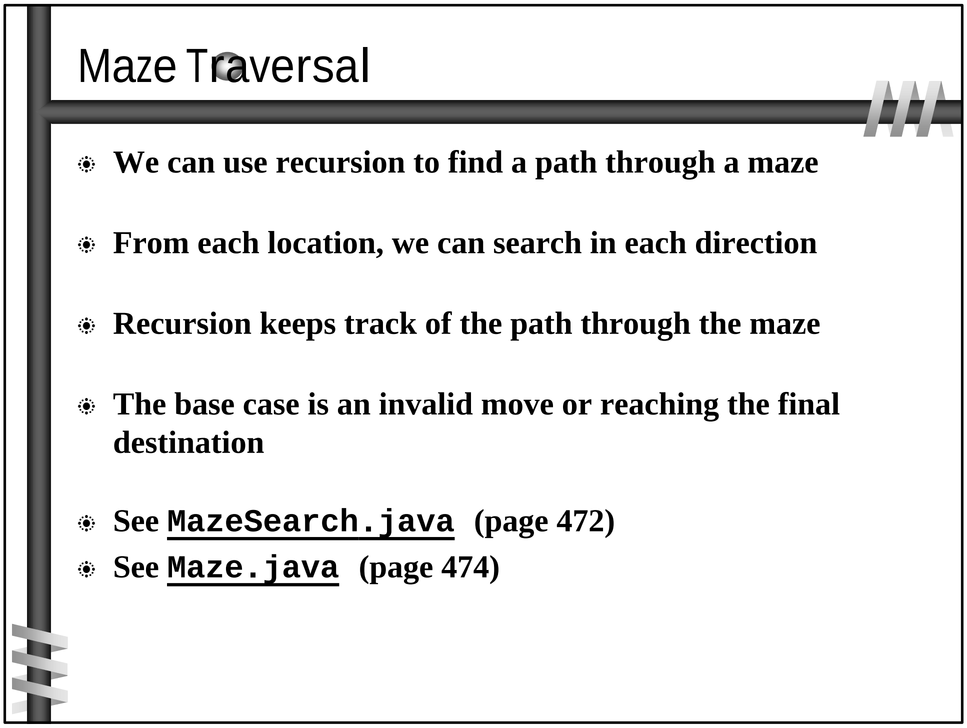
- Note that just because we can use recursion to solve a problem, doesn't mean we should
- For instance, we usually would not use recursion to solve the sum of 1 to N problem, because the iterative version is easier to understand
- However, for some problems, recursion provides an elegant solution, often cleaner than an iterative version
- You must carefully decide whether recursion is the correct technique for any problem

Indirect Recursion

- **A method invoking itself is considered to be *direct recursion***
- **A method could invoke another method, which invokes another, etc., until eventually the original method is invoked again**
- **For example, method `m1` could invoke `m2`, which invokes `m3`, which in turn invokes `m1` again**
- **This is called *indirect recursion*, and requires all the same care as direct recursion**
- **It is often more difficult to trace and debug**

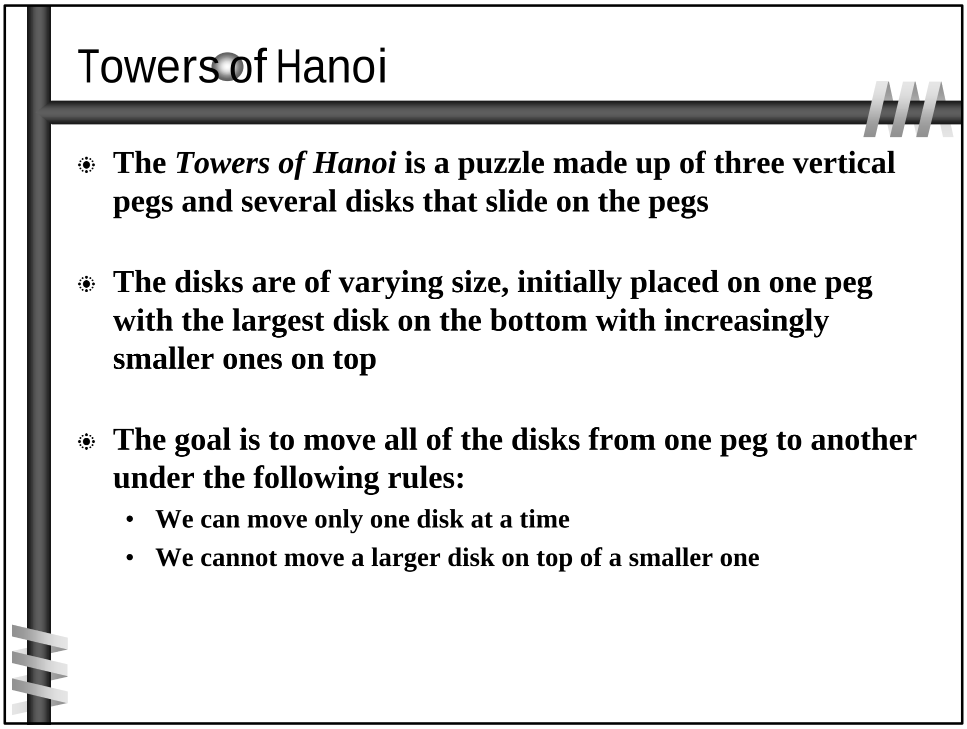
Indirect Recursion





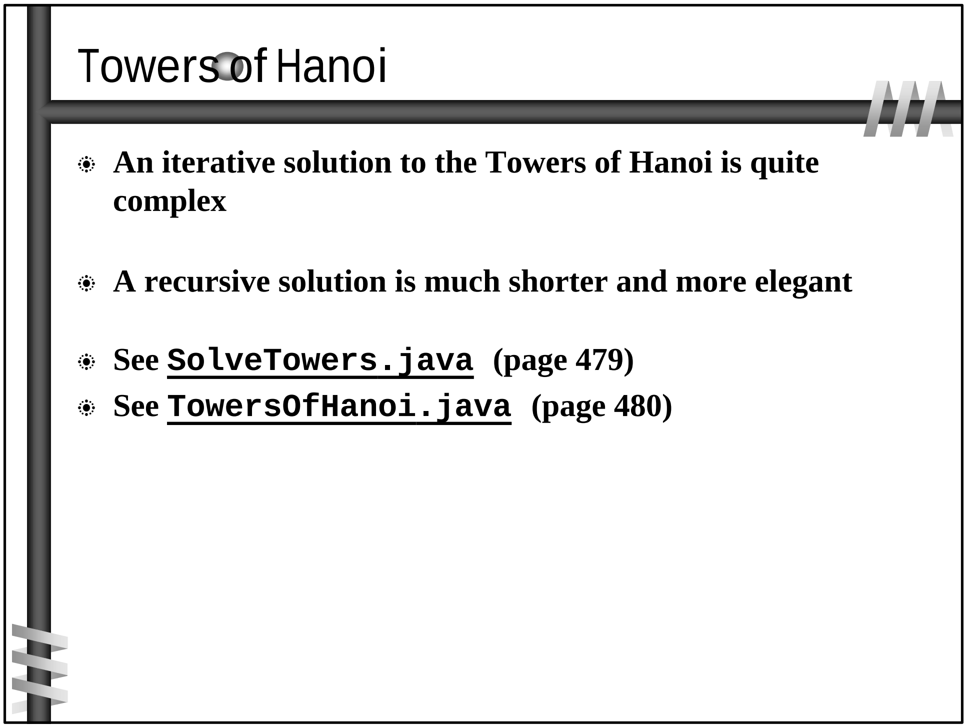
Maze Traversal

- We can use recursion to find a path through a maze
- From each location, we can search in each direction
- Recursion keeps track of the path through the maze
- The base case is an invalid move or reaching the final destination
- See `MazeSearch.java` (page 472)
- See `Maze.java` (page 474)



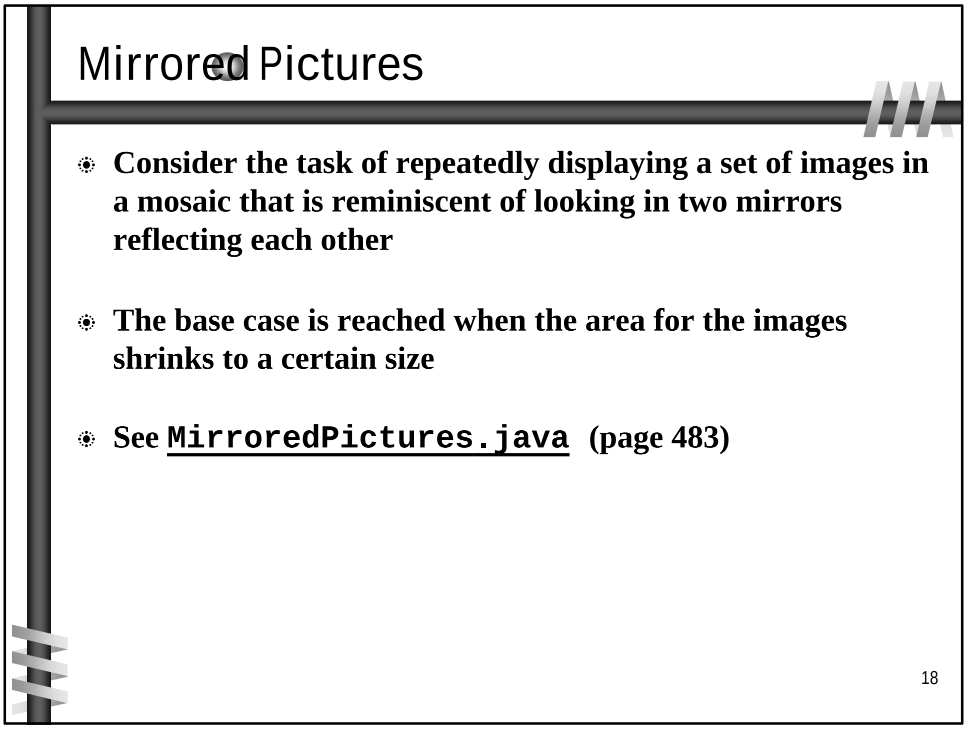
Towers of Hanoi

- The *Towers of Hanoi* is a puzzle made up of three vertical pegs and several disks that slide on the pegs
- The disks are of varying size, initially placed on one peg with the largest disk on the bottom with increasingly smaller ones on top
- The goal is to move all of the disks from one peg to another under the following rules:
  - We can move only one disk at a time
  - We cannot move a larger disk on top of a smaller one



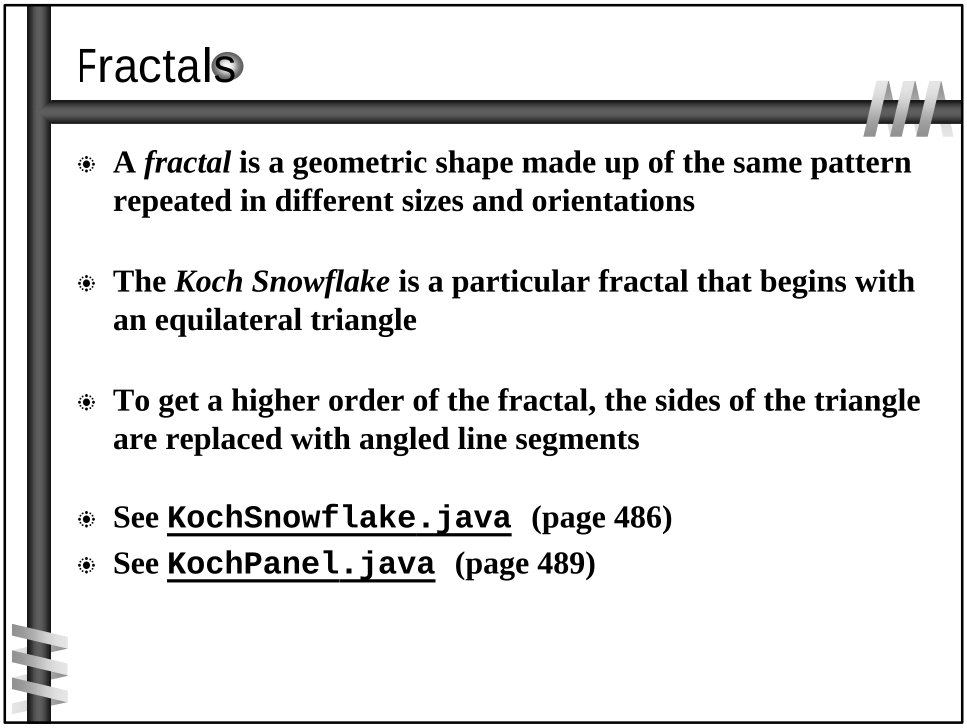
Towers of Hanoi

- An iterative solution to the Towers of Hanoi is quite complex
- A recursive solution is much shorter and more elegant
- See `SolveTowers.java` (page 479)
- See `TowersOfHanoi.java` (page 480)



Mirrored Pictures

- Consider the task of repeatedly displaying a set of images in a mosaic that is reminiscent of looking in two mirrors reflecting each other
- The base case is reached when the area for the images shrinks to a certain size
- See `MirroredPictures.java` (page 483)



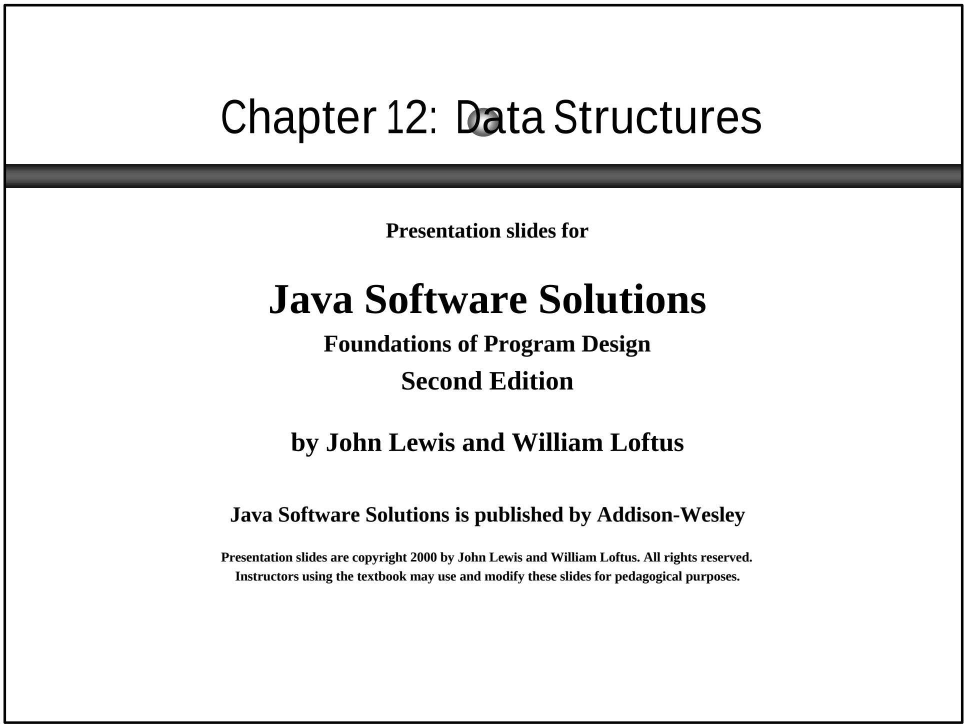
Fractals

- A *fractal* is a geometric shape made up of the same pattern repeated in different sizes and orientations

- The *Koch Snowflake* is a particular fractal that begins with an equilateral triangle

- To get a higher order of the fractal, the sides of the triangle are replaced with angled line segments

- See `KochSnowflake.java` (page 486)
- See `KochPanel.java` (page 489)



Chapter 12: Data Structures

Presentation slides for

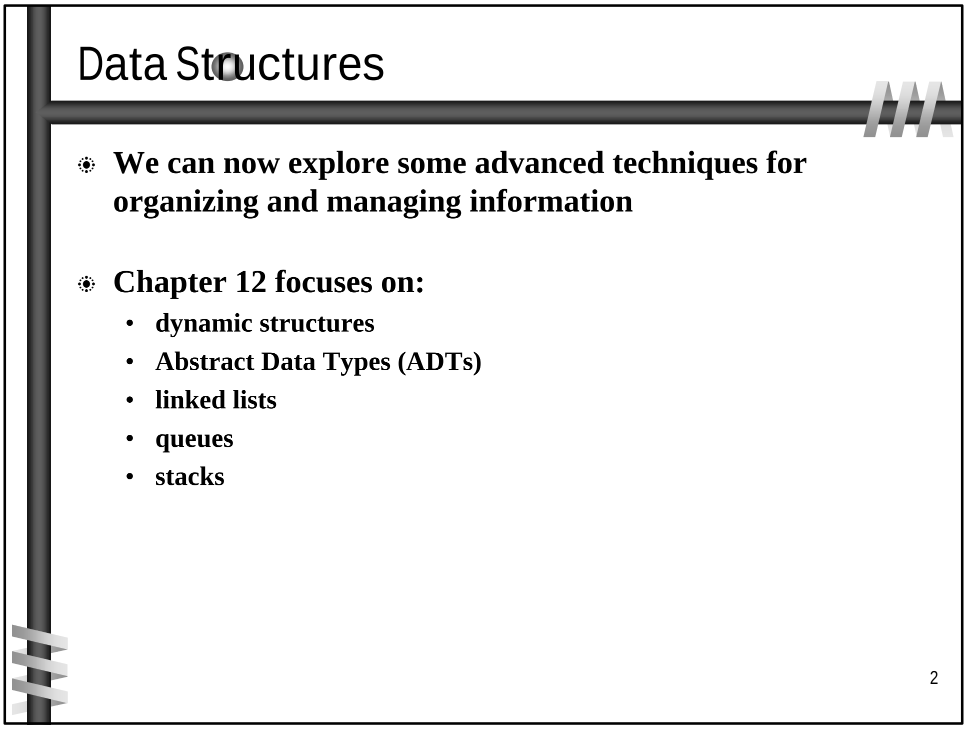
Java Software Solutions

Foundations of Program Design

Second Edition

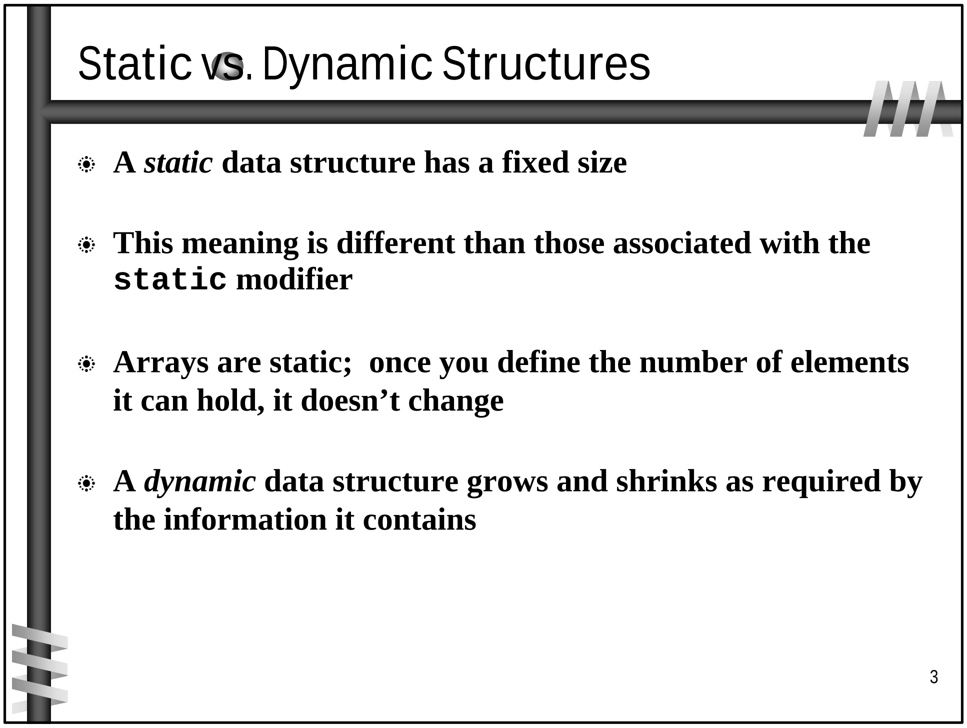
by John Lewis and William Loftus

Java Software Solutions is published by Addison-Wesley



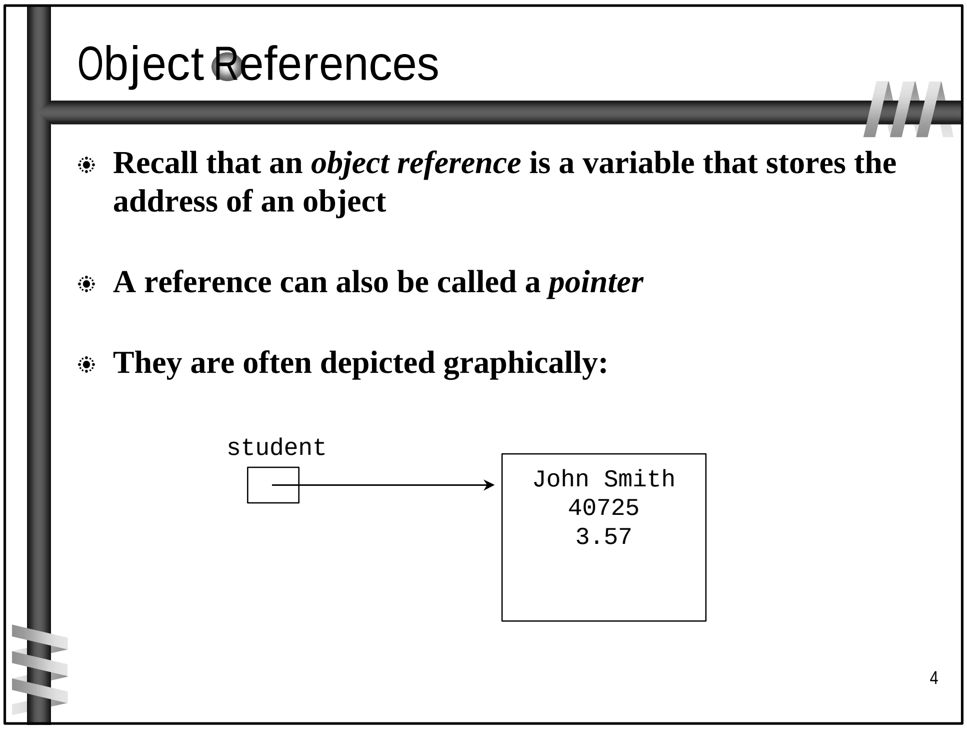
Data Structures

- **We can now explore some advanced techniques for organizing and managing information**
- **Chapter 12 focuses on:**
  - **dynamic structures**
  - **Abstract Data Types (ADTs)**
  - **linked lists**
  - **queues**
  - **stacks**



Static vs. Dynamic Structures

- **A *static* data structure has a fixed size**
- **This meaning is different than those associated with the `static` modifier**
- **Arrays are static; once you define the number of elements it can hold, it doesn't change**
- **A *dynamic* data structure grows and shrinks as required by the information it contains**

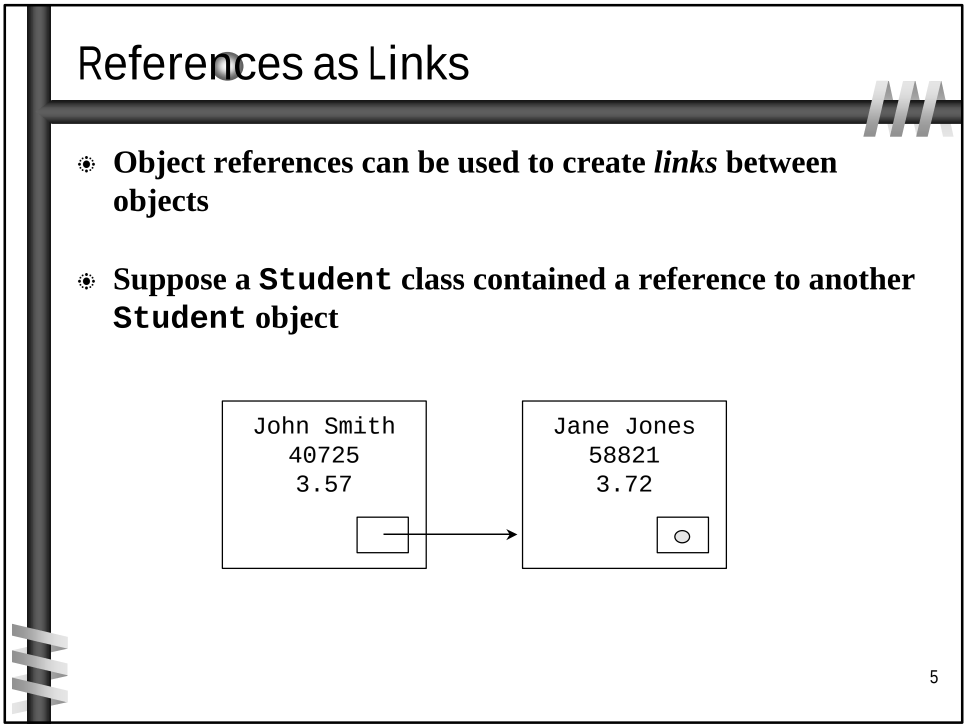


Object References

- **Recall that an *object reference* is a variable that stores the address of an object**
- **A reference can also be called a *pointer***
- **They are often depicted graphically:**

student

John Smith
40725
3.57

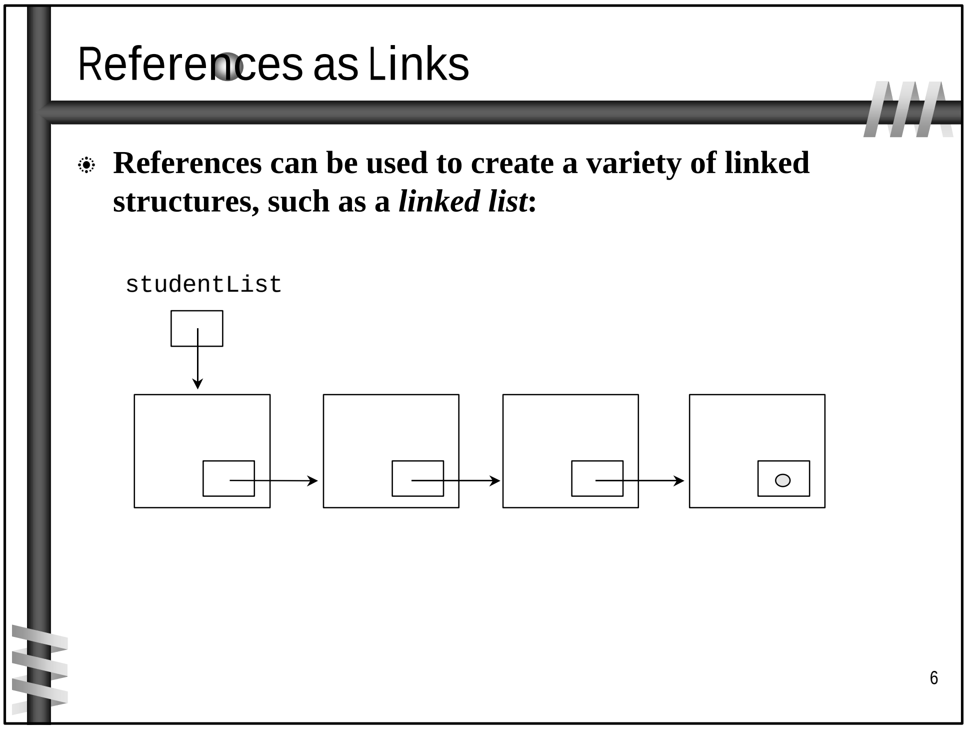


References as Links

- Object references can be used to create *links* between objects
- Suppose a `Student` class contained a reference to another `Student` object

John Smith
40725
3.57

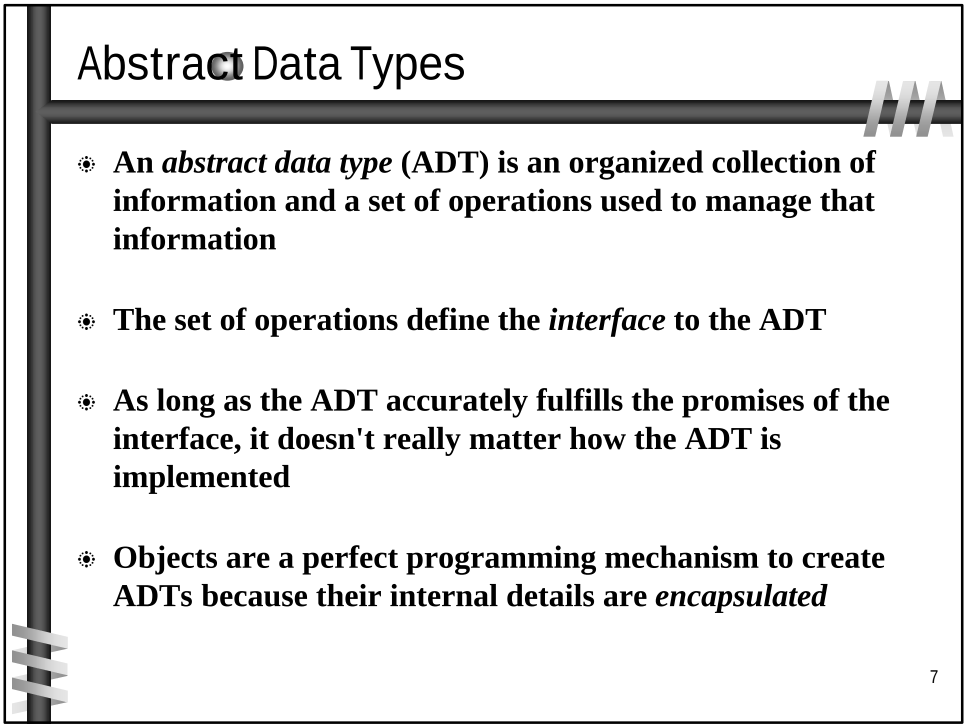
Jane Jones
58821
3.72



References as Links

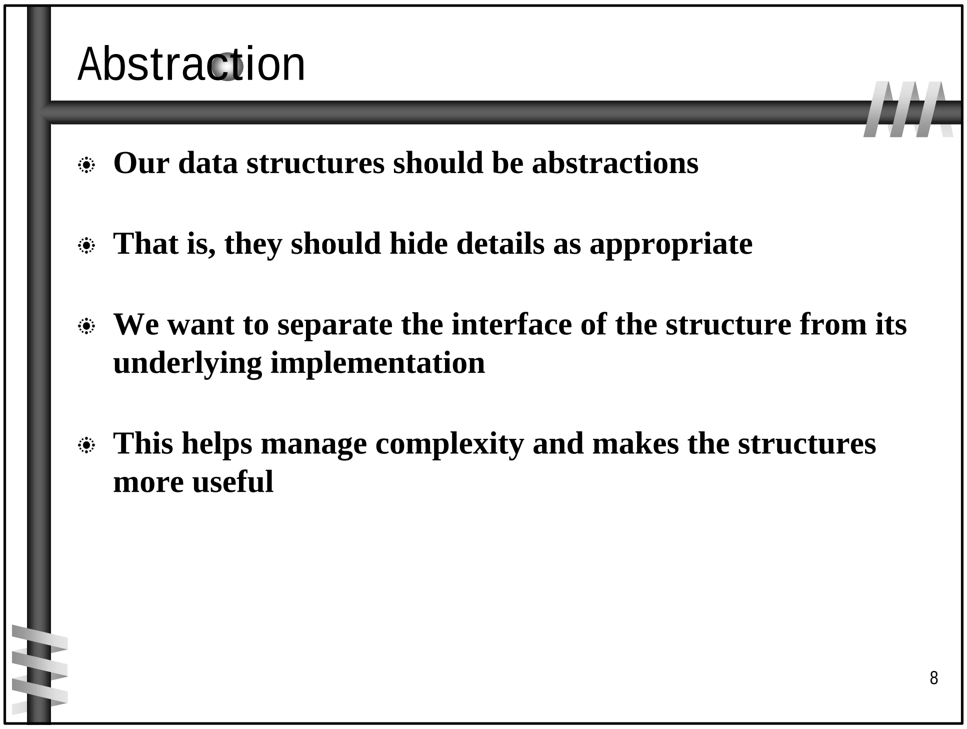
- References can be used to create a variety of linked structures, such as a *linked list*:

studentList



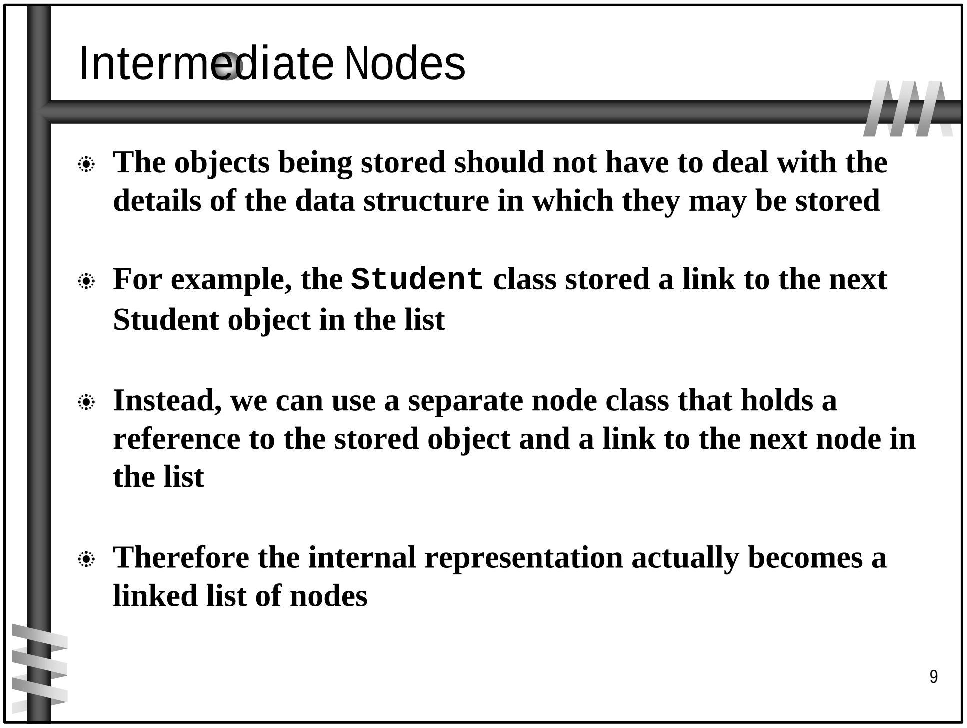
Abstract Data Types

- An *abstract data type* (ADT) is an organized collection of information and a set of operations used to manage that information
- The set of operations define the *interface* to the ADT
- As long as the ADT accurately fulfills the promises of the interface, it doesn't really matter how the ADT is implemented
- Objects are a perfect programming mechanism to create ADTs because their internal details are *encapsulated*



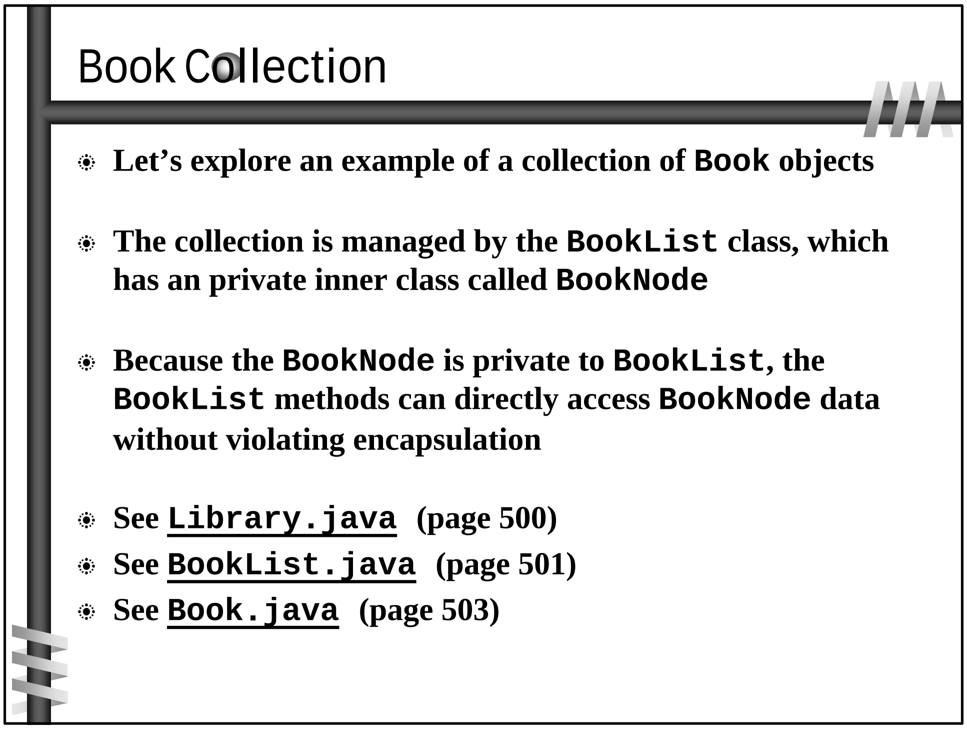
Abstraction

- Our data structures should be abstractions
- That is, they should hide details as appropriate
- We want to separate the interface of the structure from its underlying implementation
- This helps manage complexity and makes the structures more useful



Intermediate Nodes

- The objects being stored should not have to deal with the details of the data structure in which they may be stored
- For example, the `Student` class stored a link to the next Student object in the list
- Instead, we can use a separate node class that holds a reference to the stored object and a link to the next node in the list
- Therefore the internal representation actually becomes a linked list of nodes

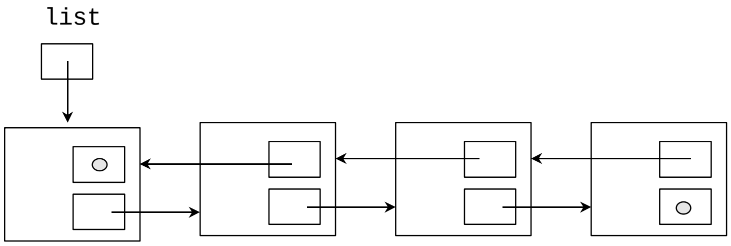


Book Collection

- Let’s explore an example of a collection of `Book` objects
- The collection is managed by the `BookList` class, which has an private inner class called `BookNode`
- Because the `BookNode` is private to `BookList`, the `BookList` methods can directly access `BookNode` data without violating encapsulation
- See `Library.java` (page 500)
- See `BookList.java` (page 501)
- See `Book.java` (page 503)

Other Dynamic List Implementations

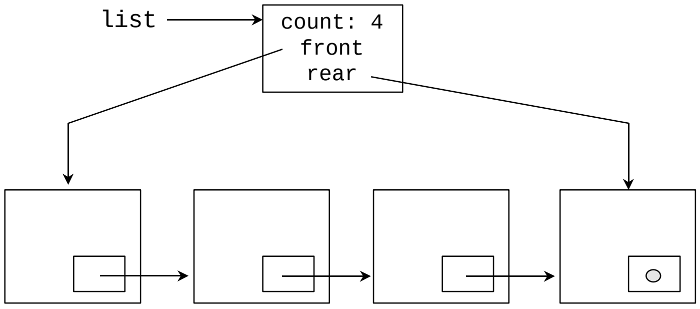
- It may be convenient to implement as list as a *doubly linked list*, with next and previous references:



list

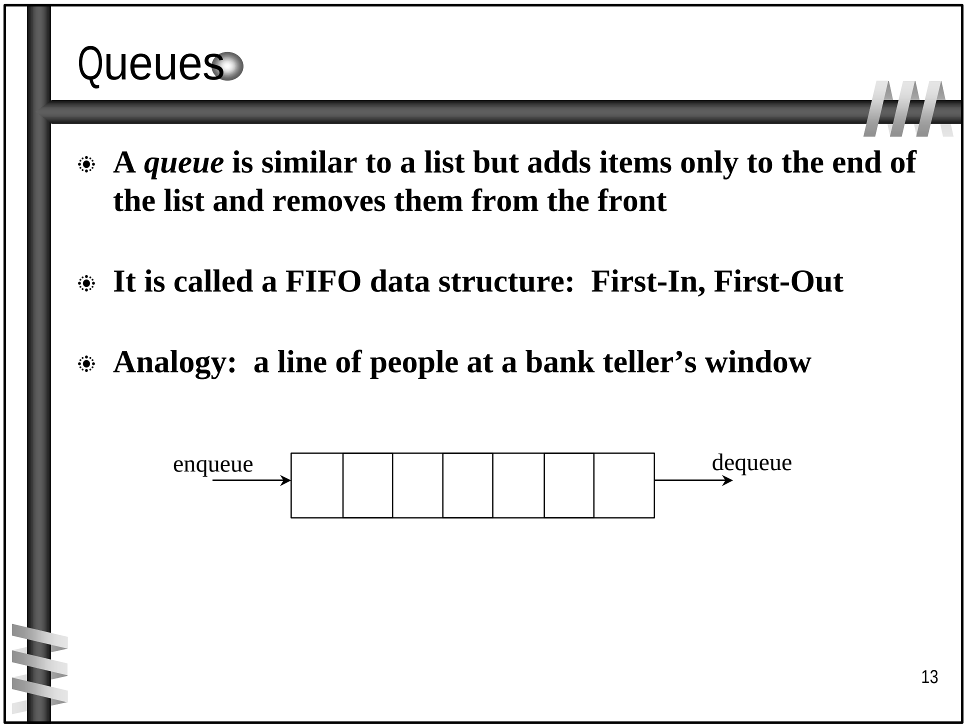
Other Dynamic List Implementations

- It may also be convenient to use a separate header node, with references to both the front and rear of the list



list

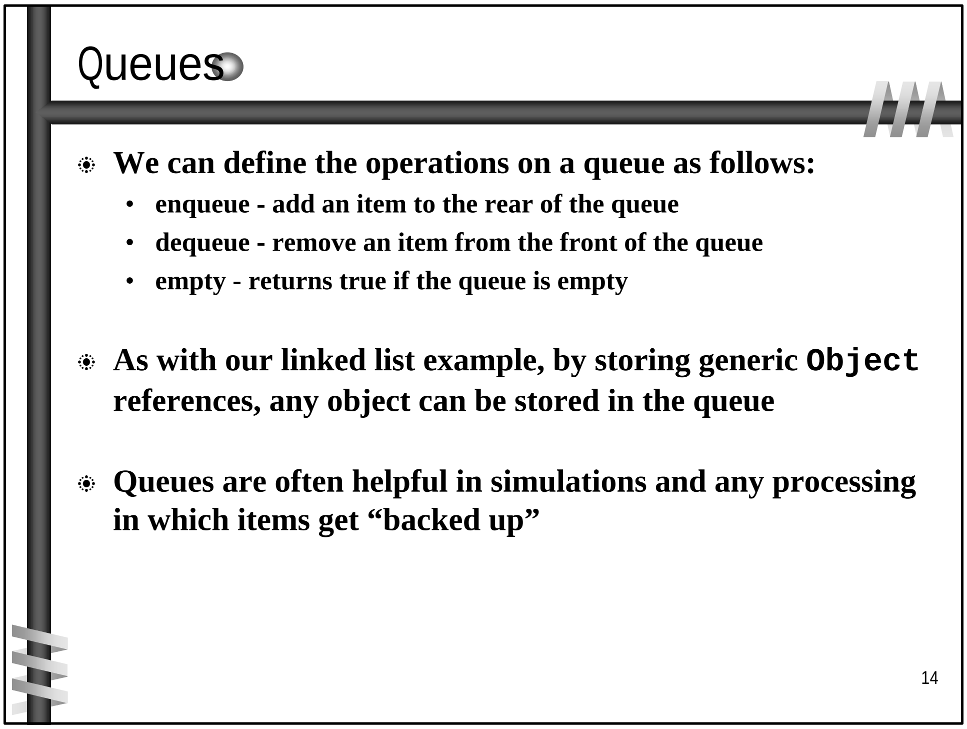
count: 4
front
rear



Queues

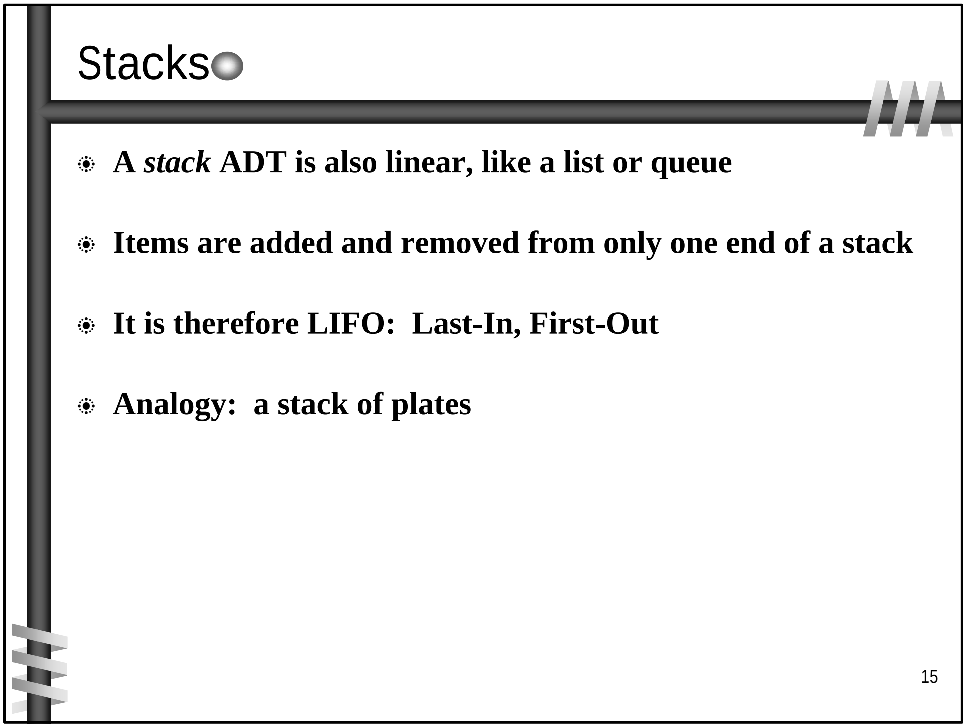
- A *queue* is similar to a list but adds items only to the end of the list and removes them from the front
- It is called a FIFO data structure: First-In, First-Out
- Analogy: a line of people at a bank teller's window

enqueue
dequeue



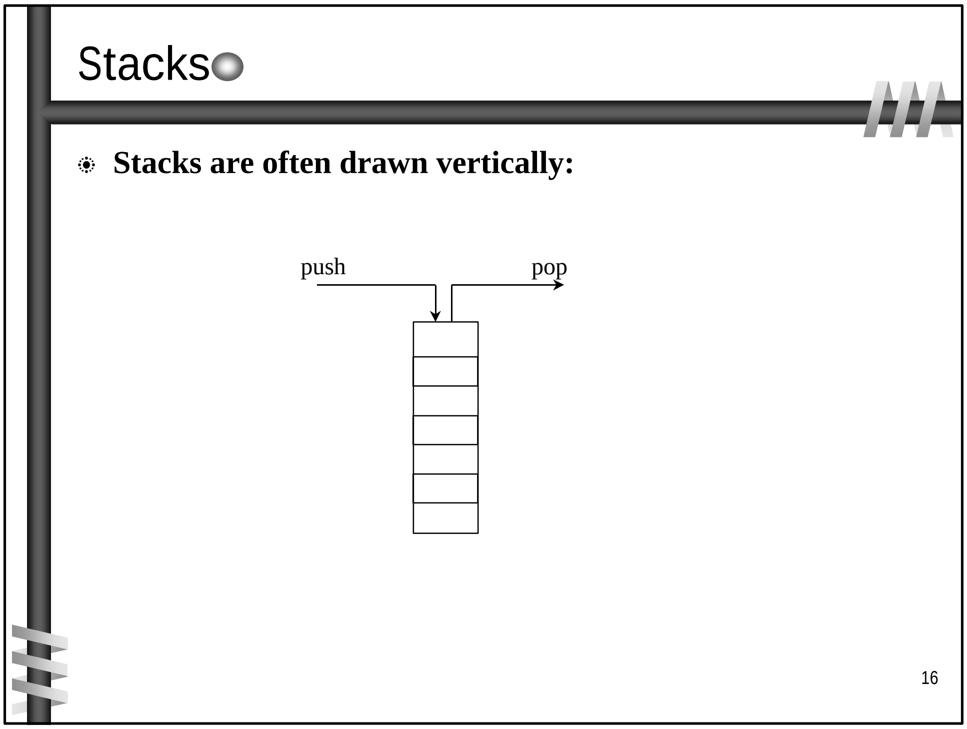
Queues

- We can define the operations on a queue as follows:
  - enqueue - add an item to the rear of the queue
  - dequeue - remove an item from the front of the queue
  - empty - returns true if the queue is empty
- As with our linked list example, by storing generic `Object` references, any object can be stored in the queue
- Queues are often helpful in simulations and any processing in which items get "backed up"



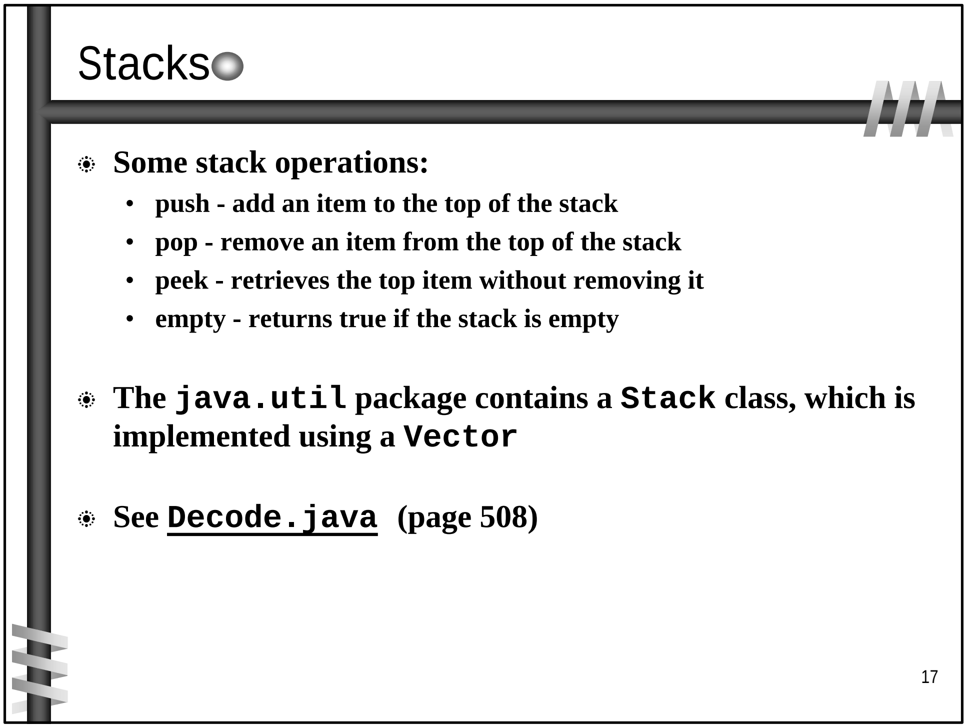
Stacks

- A *stack* ADT is also linear, like a list or queue
- Items are added and removed from only one end of a stack
- It is therefore LIFO: Last-In, First-Out
- Analogy: a stack of plates



Stacks

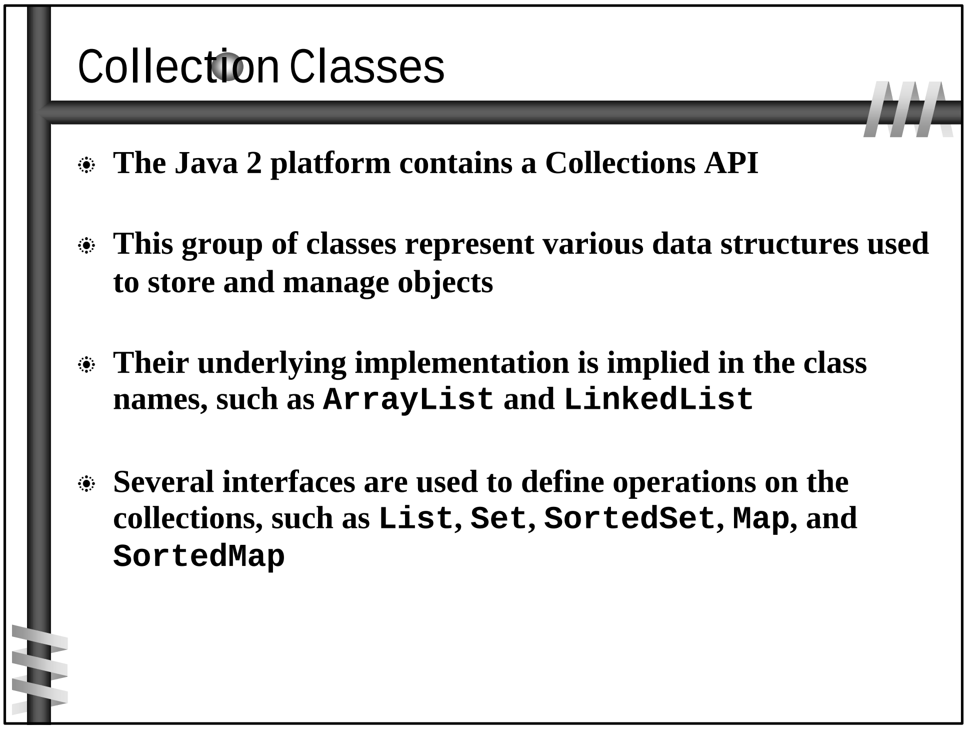
- Stacks are often drawn vertically:



Stacks

- Some stack operations:
  - push - add an item to the top of the stack
  - pop - remove an item from the top of the stack
  - peek - retrieves the top item without removing it
  - empty - returns true if the stack is empty
- The `java.util` package contains a `Stack` class, which is implemented using a `Vector`
- See `Decode.java` (page 508)

17



Collection Classes

- The Java 2 platform contains a Collections API
- This group of classes represent various data structures used to store and manage objects
- Their underlying implementation is implied in the class names, such as `ArrayList` and `LinkedList`
- Several interfaces are used to define operations on the collections, such as `List`, `Set`, `SortedSet`, `Map`, and `SortedMap`