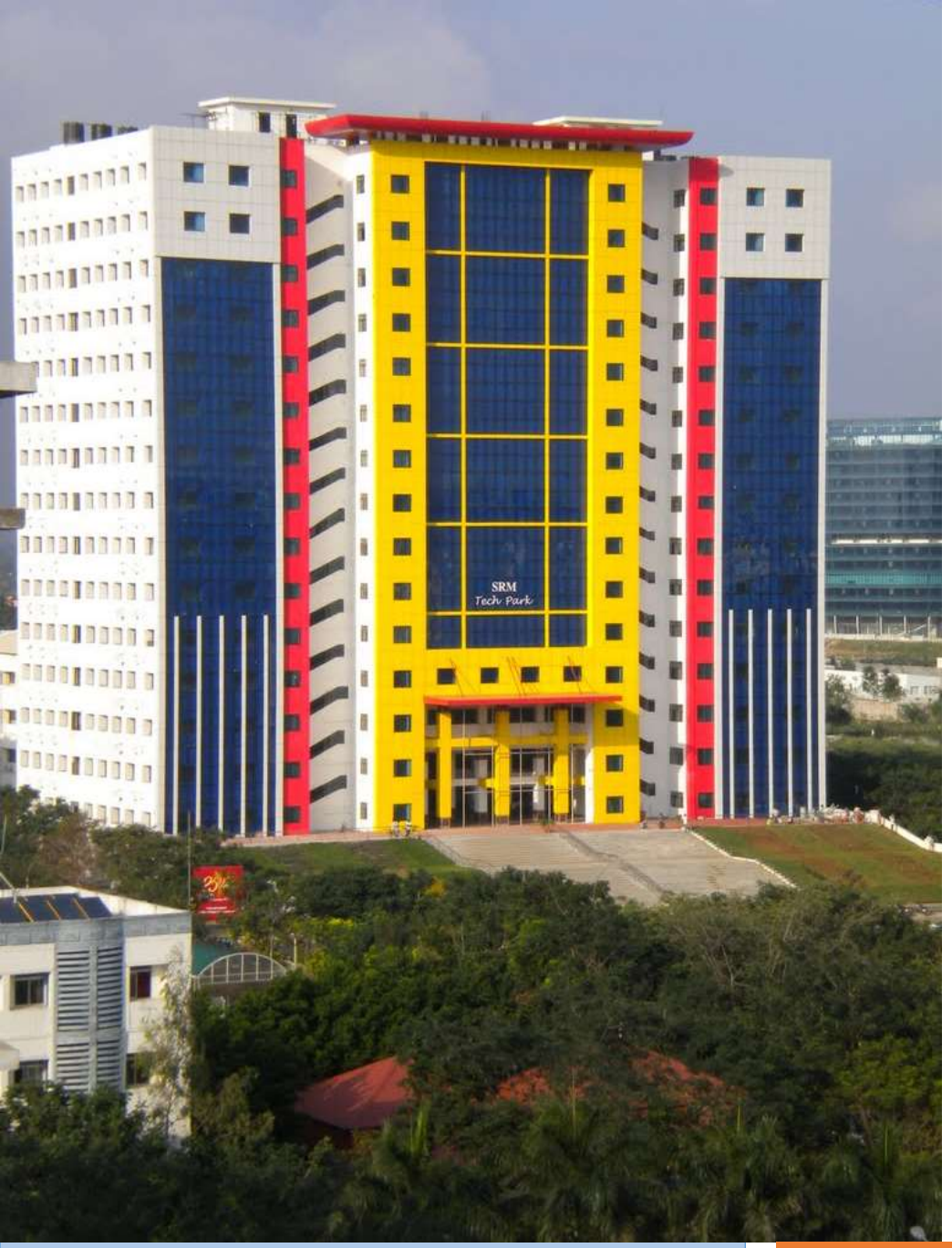


21CSC203P -Advanced Programming Practice

Unit II

Outline of the Presentation

- Object and Classes; Constructor; Data types; Variables; Modifier and Operators
- Structural Programming Paradigm: Branching, Iteration, Decision making, and Arrays
- Procedural Programming Paradigm: Characteristics; Function Definition; Function Declaration and Calling; Function Arguments
- Object-Oriented Programming Paradigm: Abstraction; Encapsulation; Inheritance; Polymorphism; Overriding
- Interfaces: Declaring, Implementing; Extended and Tagging
- Package: Package Creation

JAVA PLATFORM FEATURES

1. Compiled and Interpreter:- has both Compiled and Interpreter Feature Program of java is First Compiled and Then it is must to Interpret it. First of all The Program of java is Compiled then after Compilation it creates Bytes Codes rather than Machine Language. Then After Bytes Codes are Converted into the Machine Language is Converted into the Machine Language with the help of the Interpreter So For Executing the java Program First of all it is necessary to Compile it then it must be Interpreter
2. Platform Independent:- Java Language is Platform Independent means program of java is Easily transferable because after Compilation of java program bytes code will be created then we have to just transfer the Code of Byte Code to another Computer. This is not necessary for computers having same Operating System in which the code of the java is Created and Executed After Compilation of the Java Program We easily Convert the Program of the java top the another Computer for Execution.
3. Object-Oriented:- We Know that is purely OOP Language that is all the Code of the java Language is Written into the classes and Objects So For This feature java is Most Popular Language because it also Supports Code Reusability, Maintainability etc.

JAVA PLATFORM FEATURES

4. Robust and Secure:- The Code of java is Robust and Means of first checks the reliability of the code before Execution When We trying to Convert the Higher data type into the Lower Then it Checks the Demotion of the Code the It Will Warns a User to Not to do this So it is called as Robust. Secure : When We convert the Code from One Machine to Another the First Check the Code either it is Effected by the Virus or not or it Checks the Safety of the Code if code contains the Virus then it will never Executed that code on to the Machine.
5. Distributed:- Java is Distributed Language Means because the program of java is compiled onto one machine can be easily transferred to machine and Executes them on another machine because facility of Bytes Codes So java is Specially designed For Internet Users which uses the Remote Computers For Executing their Programs on local machine after transferring the Programs from Remote Computers or either from the internet.
6. Simple Small and Familiar:- is a simple Language Because it contains many features of other Languages like c and C++ and Java Removes Complexity because it doesn't use pointers, Storage Classes and Go to Statements and java Doesn't support Multiple Inheritance
7. Multithreaded and Interactive:- Java uses Multithreaded Techniques For Execution Means Like in other in Structure Languages Code is Divided into the Small Parts Like These Code of java is divided into the Smaller parts those are Executed by java in Sequence and Timing Manner this is Called as Multithreaded In this Program of java is divided into the Small parts those are Executed by Compiler of java itself Java is Called as Interactive because Code of java Sup
8. Dynamic and Extensible Code:- Java has Dynamic and Extensible Code Means With the Help of OOPS java Provides Inheritance and With the Help of Inheritance we Reuse the Code that is Pre-defined and Also uses all the built in Functions of java and Classes

JAVA PLATFORM FEATURES

9. Distributed:- Java is a distributed language which means that the program can be design to run on computer networks. Java provides an extensive library of classes for communicating ,using TCP/IP protocols such as HTTP and FTP. This makes creating network connections much easier than in C/C++. You can read and write objects on the remote sites via URL with the same ease that programmers are used to when read and write data from and to a file. This helps the programmers at remote locations to work together on the same project.
10. Secure: Java was designed with security in mind. As Java is intended to be used in networked/distributor environments so it implements several security mechanisms to protect you against malicious code that might try to invade your file system. For example: The absence of pointers in Java makes it impossible for applications to gain access to memory locations without proper authorization as memory allocation and referencing model is completely opaque to the programmer and controlled entirely by the underlying run-time platform .
11. Architectural Neutral: One of the key feature of Java that makes it different from other programming languages is architectural neutral (or platform independent). This means that the programs written on one platform can run on any other platform without having to rewrite or recompile them. In other words, it follows 'Write-once-run-anywhere' approach. Java programs are compiled into byte-code format which does not depend on any machine architecture but can be easily translated into a specific machine by a Java Virtual Machine (JVM) for that machine. This is a significant advantage when developing applets or applications that are downloaded from the Internet and are needed to run on different systems.

JAVA PLATFORM FEATURES

12. Portable : The portability actually comes from architecture-neutrality. In C/C++, source code may run slightly differently on different hardware platforms because of how these platforms implement arithmetic operations. In Java, it has been simplified. Unlike C/C++, in Java the size of the primitive data types are machine independent. For example, an int in Java is always a 32-bit integer, and float is always a 32-bit IEEE 754 floating point number. These consistencies make Java programs portable among different platforms such as Windows, Unix and Mac .
13. Interpreted : Unlike most of the programming languages which are either complied or interpreted, Java is both complied and interpreted The Java compiler translates a java source file to bytecodes and the Java interpreter executes the translated byte codes directly on the system that implements the Java Virtual Machine. These two steps of compilation and interpretation allow extensive code checking and improved security .
14. High performance: Java programs are complied to portable intermediate form know as bytecodes, rather than to native machine level instructions and JVM executes Java bytecode on. Any machine on which it is installed. This architecture means that Java programs are faster than program or scripts written in purely interpreted languages but slower than C and C++ programs that compiled to native machine languages. Although in the early releases of Java, the interpretation of by bytecode resulted in slow performance but the advance version of JVM uses the adaptive and Just in time (JIT) compilation technique that improves performance by converting Java bytecodes to native machine instructions on the fly.

Using Methods, classes, and Objects

- Methods are similar to procedures, functions, or subroutines
- Statements within a method execute only when the method is called
- To execute a method, you call it from another method
  - “The calling method makes a method call”

Simple methods....

- Don’t require any data items (arguments or parameters), nor do they return any data items back
- You can create a method once and use it many times in different contexts

Simple Java Program

- A class to display a simple message:

```
class MyProgram
{
public static void main(String[] args)
 {
     System.out.println(“First Java program.");
}
}
```

What is an Object?

- Real world objects are things that have:

  1) state

  2) behavior

  Example: your dog:

- state – name, color, breed, sits?, barks?, wages tail?, runs?
- behavior – sitting, barking, waging tail, running
- A software object is a bundle of variables (state) and methods (operations).

What is a Class?

- A class is a blueprint that defines the variables and methods common to all objects of a certain kind.
- Example: ‘your dog’ is a object of the class Dog.
- An object holds values for the variables defines in the class.
- An object is called an instance of the Class

Object Creation

- A variable is declared to refer to the objects of type/class String:

  String s;

- The value of s is null; it does not yet refer to any object.
- A new String object is created in memory with initial “abc” value:
- String s = new String(“abc”);
- Now s contains the address of this new object.

Object Destruction

- A program accumulates memory through its execution.
- Two mechanism to free memory that is no longer need by the program:
    1) manual – done in C/C++
    2) automatic – done in Java
- In Java, when an object is no longer accessible through any variable, it is eventually removed from the memory by the garbage collector.
- Garbage collector is parts of the Java Run-Time Environment.

Class

- A basis for the Java language.
- Each concept we wish to describe in Java must be included inside a class.
- A class defines a new data type, whose values are objects:
- A class is a template for objects
- An object is an instance of a class

Class Definition

- A class contains a name, several variable declarations (instance variables) and several method declarations. All are called members of the class.
- General form of a class:

```
class classname {
        type instance-variable-1;
        …
        type instance-variable-n;
        type method-name-1(parameter-list) { … }
        type method-name-2(parameter-list) { … }
        …
        type method-name-m(parameter-list) { … }
            }
```

Example: Class Usage

```
class Box {
    double width;
    double height;
    double depth;
}
class BoxDemo {
    public static void main(String args[]) {
            Box mybox = new Box();
           double vol;
           mybox.width = 10;
           mybox.height = 20;
           mybox.depth = 15;
           vol = mybox.width * mybox.height * mybox.depth;
           System.out.println ("Volume is " + vol);
} }
```

Constructor

- A constructor initializes the instance variables of an object.
- It is called immediately after the object is created but before the new operator completes.

    1) it is syntactically similar to a method:

    2) it has the same name as the name of its class

    3) it is written without return type; the default return type of a class

- constructor is the same class
- When the class has no constructor, the default constructor automatically initializes all its instance variables with zero.

Example: Constructor

```
class Box {
    double width;
    double height;
    double depth;
Box() {
      System.out.println("Constructing Box");
      width = 10; height = 10; depth = 10;
}
double volume() {
    return width * height * depth;
}
}
```

Parameterized Constructor

```
class Box {
double width;
double height;
double depth;
Box(double w, double h, double d) {
width = w; height = h; depth = d;
}
double volume()
 { return width * height * depth;
}
}
```

KEYWORD THIS

- Can be used by any object to refer to itself in any class method
- Typically used to
  - Avoid variable name collisions
  - Pass the receiver as an argument
  - Chain constructors

KEYWORD THIS

- Keyword this allows a method to refer to the object that invoked it.
- It can be used inside any method to refer to the current object:

```
Box(double width, double height, double depth) {
this.width = width;
this.height = height;
this.depth = depth;
}
```

VISIBILITY

```
public class Circle {
    private double x,y,r;

        // Constructor
      public Circle (double x, double y, double r) {
                this.x = x;
                this.y = y;
                this.r = r;
        }
      //Methods to return circumference and area
     public double circumference() { return 2*3.14*r;}
     public double area() { return 3.14 * r * r; }
}
```

Methods

- General form of a method definition:

```
type name(parameter-list) {
                … return value;
    …
}
```

- Components:

  1) type - type of values returned by the method. If a method does not return any value, its return type must be void.

  2) name is the name of the method

  3) parameter-list is a sequence of type-identifier lists separated by commas

  4) return value indicates what value is returned by the method.

Example: Method

- Classes declare methods to hide their internal data structures, as well as for their own internal use: Within a class, we can refer directly to its member variables:

```
class Box {
double width, height, depth;
void volume() {
System.out.print("Volume is ");
System.out.println(width * height * depth);
}
}
```

Parameterized Method

- Parameters increase generality and applicability of a method:
- 1) method without parameters

```
int square() { return 10*10; }
```

- 2) method with parameters

```
int square(int i) { return i*i; }
```

- Parameter: a variable receiving value at the time the method is invoked.
- Argument: a value passed to the method when it is invoked.

Access Control: Data Hiding and Encapsulation

- Java provides control over the *visibility* of variables and methods.
- *Encapsulation*, safely sealing data within the *capsule* of the class Prevents programmers from relying on details of class implementation, so you can update without worry
- Helps in protecting against accidental or wrong usage.
- Keeps code elegant and clean (easier to maintain)

Access Modifiers: Public, Private, Protected

- *Public:* keyword applied to a class, makes it available/visible everywhere. Applied to a method or variable, completely visible.
- Default(No visibility modifier is specified): it behaves like public in its package and private in other packages.
- *Default Public* keyword applied to a class, makes it available/visible everywhere. Applied to a method or variable, completely visible.
- *Private* fields or methods for a class only visible within that class. Private members are *not* visible within subclasses, and are *not* inherited.
- *Protected* members of a class are visible within the class, subclasses and *also* within all classes that are in the same package as that class.

Expressions in Java

- The heart of the Add2Integers program from Chapter 2 is the line

```
int total = n1 + n2;
```

that performs the actual addition.

- The n1 + n2 that appears to the right of the equal sign is an example of an expression, which specifies the operations involved in the computation.
- An expression in Java consists of terms joined together by operators.
- Each term must be one of the following:
  - A constant (such as 3.14159265 or "hello, world")
  - A variable name (such as n1, n2, or total)
  - A method calls that returns a values (such as readInt)
  - An expression enclosed in parentheses

Primitive Data Types

- Although complex data values are represented using objects, Java defines a set of primitive types to represent simple data.
- Of the eight primitive types available in Java, the programs in this text use only the following four:

`int`	This type is used to represent integers, which are whole numbers such as 17 or −53.
`double`	This type is used to represent numbers that include a decimal fraction, such as 3.14159265. In Java, such values are called floating-point numbers; the name `double` comes from the fact that the representation uses twice the minimum precision.
`boolean`	This type represents a logical value (`true` or `false`).
`char`	This type represents a single character and is described in Chapter 8.

Summary of the Primitive Types

A data type is defined by a set of values called the domain and a set of operations.

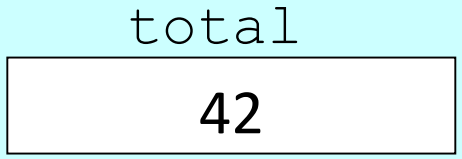
The following table shows the data domains and common operations for all eight of Java's primitive types:

Type	*Domain*	*Common operations*
`byte`	8-bit integers in the range –128 to 127	*The arithmetic operators:* `+` add, `-` subtract, `*` multiply, `/` divide, `%` remainder *The relational operators:* `==` equal to, `!=` not equal, `<` less than, `<=` less or equal, `>` greater than, `>=` greater or equal
`short`	16-bit integers in the range –32768 to 32767	
`int`	32-bit integers in the range –2146483648 to 2146483647	
`long`	64-bit integers in the range –9223372036754775808 to 9223372036754775807	
`float`	32-bit floating-point numbers in the range $\pm 1.4 \times 10^{-45}$ to $\pm 3.4028235 \times 10^{-38}$	*The arithmetic operators except* `%` *The relational operators*
`double`	64-bit floating-point numbers in the range $\pm 4.39 \times 10^{-322}$ to $\pm 1.7976931348623157 \times 10^{308}$	
`char`	16-bit characters encoded using Unicode	*The relational operators*
`boolean`	the values `true` and `false`	*The logical operators:* `&&` add, `||` or, `!` not

Constants and Variables

- The simplest terms that appear in expressions are constants and variables. The value of a constant does not change during the course of a program. A variable is a placeholder for a value that can be updated as the program runs.

- A variable in Java is most easily envisioned as a box capable of storing a value.



(contains an `int`)

- Each variable has the following attributes:
  - A name, which enables you to differentiate one variable from another.
  - A type, which specifies what type of value the variable can contain.
  - A value, which represents the current contents of the variable.
- The name and type of a variable are fixed. The value changes whenever you assign a new value to the variable.

Java Identifiers

- Names for variables (and other things) are called identifiers.
- Identifiers in Java conform to the following rules:
  - A variable name must begin with a letter or the underscore character.
  - The remaining characters must be letters, digits, or underscores.
  - The name must not be one of Java's reserved words:

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

  - Identifiers should make their purpose obvious to the reader.
  - Identifiers should adhere to standard conventions. Variable names, for example, should begin

Variable Declarations

- In Java, you must declare a variable before you can use it. The declaration establishes the name and type of the variable and, in most cases, specifies the initial value as well.

- The most common form of a variable declaration is

```
type name = value;
```

where *type* is the name of a Java primitive type or class, *name* is an identifier that indicates the name of the variable, and *value* is an expression specifying the initial value.

- Most declarations appear as statements in the body of a method definition. Variables declared in this way are called local variables and are accessible only inside that method.
- Variables may also be declared as part of a class. These are called instance variables and are covered in Chapter 6.

Operators and Operands

- As in most languages, Java programs specify computation in the form of arithmetic expressions that closely resemble expressions in mathematics.

- The most common operators in Java are the ones that specify arithmetic computation:

+	Addition	*	Multiplication
−	Subtraction	/	Division
		%	Remainder

- Operators in Java usually appear between two subexpressions, which are called its operands. Operators that take two operands are called binary operators.

- The - operator can also appear as a unary operator, as in the expression -x, which denotes the negative of x.

```
package conversion;
public class Main
{
    public static void main(String[] args)
    {
        int var1=12;
        double var2=45.8921;
        double result=var1+var2;
        char letter=65;
        System.out.println("The sum gives "+result);
        System.out.println("The letter is "+letter);
    }
}
```

```
run:
The sum gives 57.8921
The letter is A
BUILD SUCCESSFUL (total time: 0 seconds)
```

Division and Type Casts

- Whenever you apply a binary operator to numeric values in Java, the result will be of type int if both operands are of type int, but will be a double if either operand is a double.
- This rule has important consequences in the case of division. For example, the expression

```
14 / 5
```

seems as if it should have the value 2.8, but because both operands are of type int, Java computes an integer result by throwing away the fractional part. The result is therefore 2.

- If you want to obtain the mathematically correct result, you need to convert at least one operand to a double, as in

```
(double) 14 / 5
```

The conversion is accomplished by means of a type cast, which consists of a type name in parentheses.

```
package casting;

public class Main
{
    public static void main(String[] args)
    {
        double x=10.0, y=3.0;
        int i=(int)(x/y);
        System.out.println("Integer outcome was: "+i);
    }
}
```

```
run:
Integer outcome was: 3
BUILD SUCCESSFUL (total time: 1 second)
```

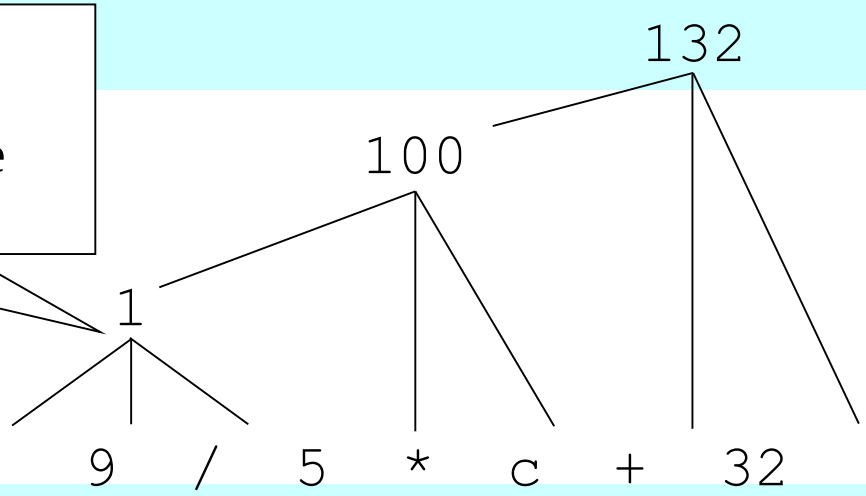
The Pitfalls of Integer Division

Consider the following Java statements, which are intended to convert 100˚ Celsius temperature to its Fahrenheit equivalent:

```
double c = 100;
double f = 9 / 5 * c + 32;
```

The computation consists of evaluating the following expression:

The problem arises from the fact that both 9 and 5 are of type int, which means that the result is also an int.

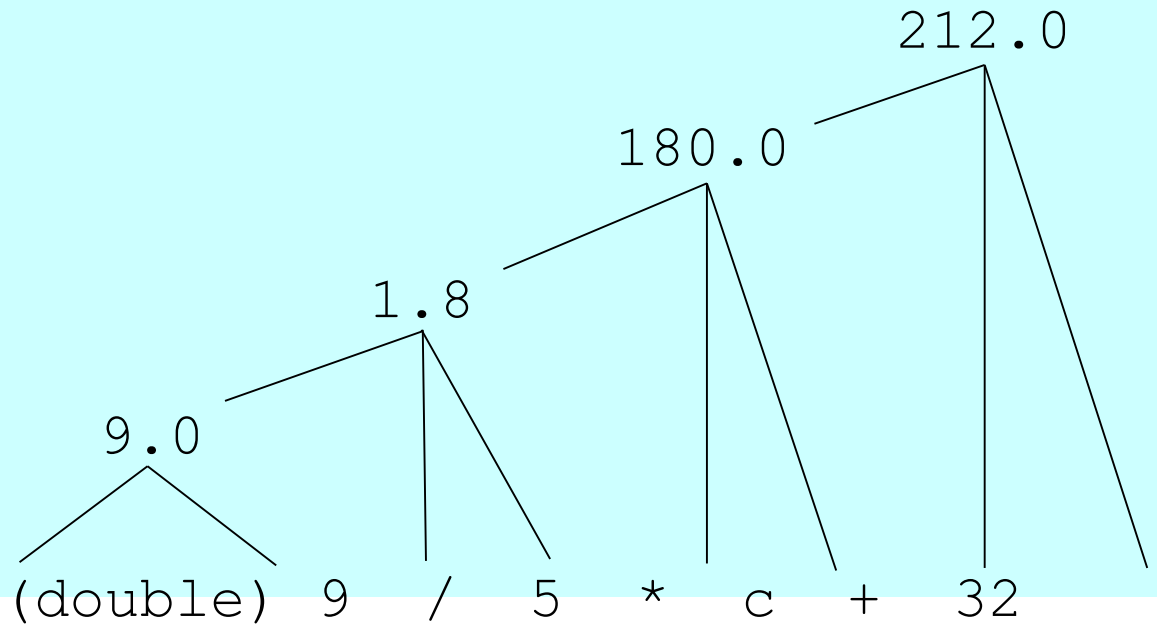


The Pitfalls of Integer Division

You can fix this problem by converting the fraction to a double, either by inserting decimal points or by using a type cast:

```
double c = 100;
double f = (double) 9 / 5 * c + 32;
```

The computation now looks like this:



The Remainder Operator

- The only arithmetic operator that has no direct mathematical counterpart is %, which applies only to integer operands and computes the remainder when the first divided by the second:

```
14 % 5    returns    4
14 % 7    returns    0
7 % 14    returns    7
```

- The result of the % operator make intuitive sense only if both operands are positive. The examples in this book do not depend on knowing how % works with negative numbers.
- The remainder operator turns out to be useful in a surprising number of programming applications and is well worth a bit of study.

Precedence

- If an expression contains more than one operator, Java uses precedence rules to determine the order of evaluation. The arithmetic operators have the following relative precedence:

unary –	(*type cast*)	
*	/	%
+	–	

highest

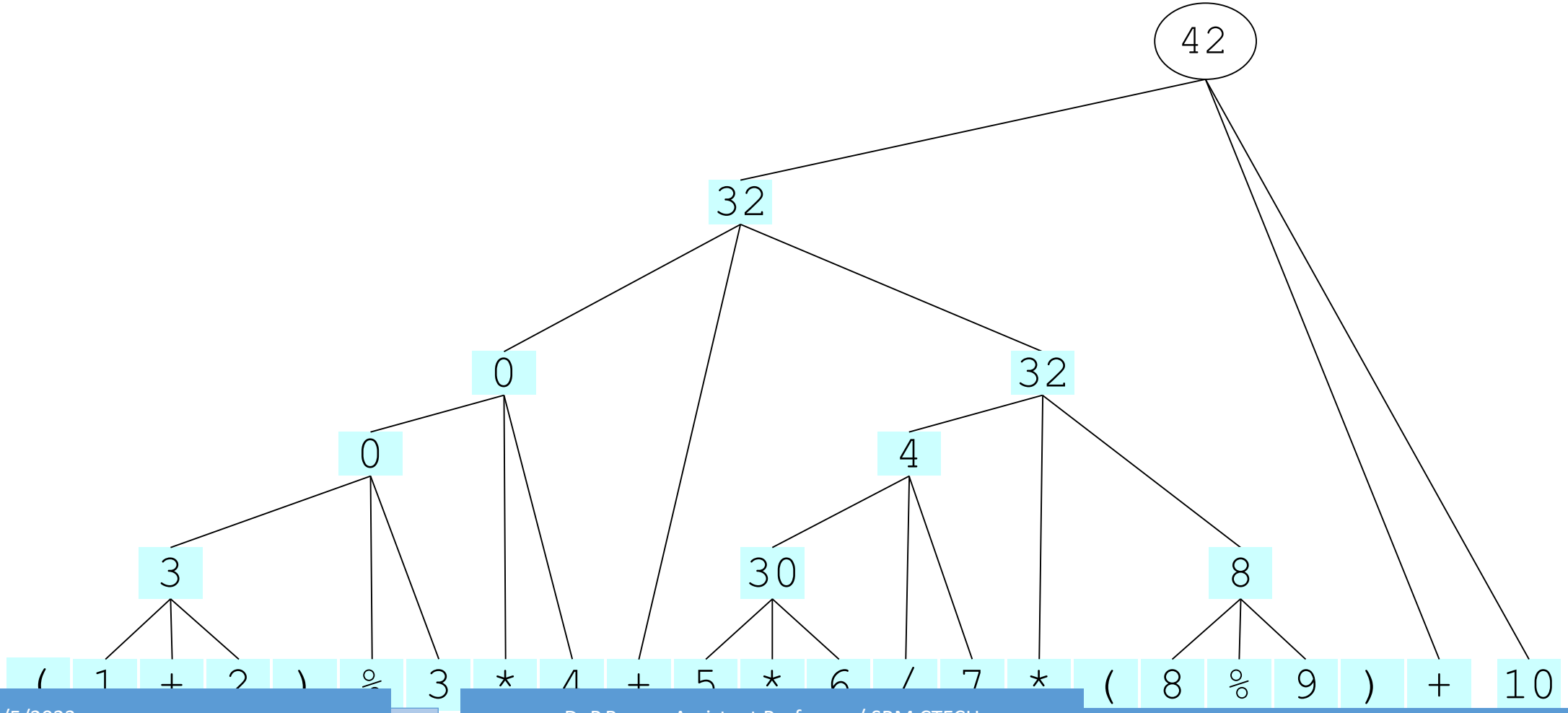
lowest

Thus, Java evaluates unary - operators and type casts first, then the operators *, /, and %, and then the operators + and -.

- Precedence applies only when two operands compete for the same operator. If the operators are independent, Java evaluates expressions from left to right.
- Parentheses may be used to change the order of operations.

Exercise: Precedence Evaluation

What is the value of the expression at the bottom of the screen?



Assignment Statements

- You can change the value of a variable in your program by using an assignment statement, which has the general form:

```
variable = expression;
```

- The effect of an assignment statement is to compute the value of the expression on the right side of the equal sign and assign that value to the variable that appears on the left. Thus, the assignment statement

```
total = total + value;
```

adds together the current values of the variables total and value and then stores that sum back in the variable total.

- When you assign a new value to a variable, the old value of that variable is lost.

Shorthand Assignments

- Statements such as

```
total = total + value;
```

are so common that Java allows the following shorthand form:

```
total += value;
```

- The general form of a shorthand assignment is

```
variable op= expression;
```

where *op* is any of Java's binary operators. The effect of this statement is the same as

```
variable = variable op (expression);
```

For example, the following statement multiplies salary by 2.

```
salary *= 2;
```

Increment and Decrement Operators

- Another important shorthand form that appears frequently in Java programs is the increment operator, which is most commonly written immediately after a variable, like this:

```
x++;
```

The effect of this statement is to add one to the value of x, which means that this statement is equivalent to

```
x += 1;
```

or in an even longer form

```
x = x + 1;
```

- The -- operator (which is called the decrement operator) is similar but subtracts one instead of adding one.
- The ++ and -- operators are more complicated than shown here, but it makes sense to defer the details until Chapter 11.

Boolean Operators

- The operators used with the boolean data type fall into two categories: relational operators and logical operators.
- There are six relational operators that compare values of other types and produce a boolean result:

= =	Equals	!=	Not equals
<	Less than	<=	Less than or equal to
>	Greater than	>=	Greater than or equal to

For example, the expression n <= 10 has the value true if x is less than or equal to 10 and the value false otherwise.

- There are also three logical operators:

&&	Logical AND	p && q means both p and q
||	Logical OR	p || q means either p or q (or both)
!	Logical NOT	!p means the opposite of p

Notes on the Boolean Operators

- Remember that Java uses = to denote assignment. To test whether two values are equal, you must use the = = operator.

- It is not legal in Java to use more than one relational operator in a single comparison as is often done in mathematics. To express the idea embodied in the mathematical expression

$$0 \le x \le 9$$

you need to make both comparisons explicit, as in

```
0 <= x && x <= 9
```

- The || operator means *either or both*, which is not always clear in the English interpretation of *or.*

- Be careful when you combine the ! operator with && and || because the interpretation often differs from informal English.

Java Comparison Operators

Math Notation	Name	Java Notation	Java Examples
=	Equal to	==	`balance == 0` `answer == 'y'`
≠	Not equal to	!=	`income != tax` `answer != 'y'`
>	Greater than	>	`expenses > income`
≥	Greater than or equal to	>=	`points >= 60`
<	Less than	<	`pressure < max`
≤	Less than or equal to	<=	`expenses <= income`

Display 3.2

Java Comparison Operators

Compound Boolean Expressions

- Boolean expressions can be combined using the “and” (&&) operator.
- example

```
if ((score > 0) && (score <= 100))
...
```

- not allowed

```
if (0 < score <= 100)
...
```

Compound Boolean Expressions, cont.

- syntax

  ```
  (Sub_Expression_1) && (Sub_Expression_2)
  ```

- Parentheses often are used to enhance readability.
- The larger expression is true only when both of the smaller expressions are true.

Compound Boolean Expressions, cont.

- Boolean expressions can be combined using the “or” (||) operator.
- Example

```
if ((quantity > 5) || (cost < 10))
...
```

- syntax

```
(Sub_Expression_1) || (Sub_Expression_2)
```

Compound Boolean Expressions, cont.

- The larger expression is true
  - when either of the smaller expressions is true
  - when both of the smaller expressions are true.
- "or" in Java is *inclusive or*
  - either or both to be true.
- *exclusive or*
  - one or the other, but not both to be true.

Negating a Boolean Expression

- Boolean negation
  - “not” (!) operator.
- syntax

  !*Boolean_Expression*
- Example:

  Boolean walk = false;

  System.out.println(!walk);

Truth Tables

&& (and)

Value of *A*	Value of *B*	Resulting Value of *A* && *B*
true	true	true
true	false	false
false	true	false
false	false	false

|| (or)

Value of *A*	Value of *B*	Resulting Value of *A* || *B*
true	true	true
true	false	true
false	true	true
false	false	false

! (not)

Value of *A*	Resulting Value of ! (*A*)
true	false
false	true

Display 3.15
Truth Tables for Boolean Operators

Primary Logical Operators

- Primary logical operators: and, or, not
- **Any** logical expression can be composed
- Example: *exclusive or*

  (a || b) && !(a && b)
- Either work or play:

  (work || play) && !(work && play)
- ^ is exclusive-or in Java
  - work ^ play
  - not a logical operator in most languages

Using ==

- == is appropriate for determining if two integers or characters have the same value.

  if (a == 3)

  where a is an integer type

- == is not appropriate for determining if two floating point values are equal.
  - Use < and some appropriate tolerance instead.

    if (Math.abs(b - c) < epsilon)
  - b, c, and epsilon are of floating point type

  [www.cs.fit.edu/~pkc/classes/cse1001/FloatEquality.java]

Using ==, cont.

- == is not appropriate for determining if two objects have the same value.
  - if (s1 == s2)
    - determines only if s1 and s2 are at the same memory location.
  - If s1 and s2 refer to strings with identical sequences of characters, but stored in different memory locations
    - (s1 == s2) is false.

Using ==, cont.

- To test the equality of objects of class String, use method equals.
  s1.equals(s2)
  or
  s2.equals(s1)
  www.cs.fit.edu/~pkc/classes/cse1001/StringEqual.java
- To test for equality ignoring case, use method equalsIgnoreCase.
  (“Hello”.equalsIgnoreCase(“hello”))

EQUALS AND EQUALS IGNORECASE

- syntax

  String.equals(*Other_String*)

  String.equalsIgnoreCase(*Other_String*)

Testing Strings for Equality

- `class StringEqualityDemo`

```
import java.util.*;

public class StringEqualityDemo
{
    public static void main(String[] args)
    {
        String s1, s2;

        System.out.println("Enter two lines of text:");

        Scanner keyboard = new Scanner(System.in);
        s1 = keyboard.nextLine();
        s2 = keyboard.nextLine();

        if (s1.equals(s2))
            System.out.println("The two lines are equal.");
        else
            System.out.println("The two lines are not equal.");

        if (s2.equals(s1))
            System.out.println("The two lines are equal.");
        else
            System.out.println("The two lines are not equal.");

        if (s1.equalsIgnoreCase(s2))
            System.out.println(
                        "But the lines are equal, ignoring case.");
        else
            System.out.println(
                        "Lines are not equal, even ignoring case.");
    }
}
```

These two invocations of the method `equals` *are equivalent.*

Sample Screen Dialog

```
Enter two lines of text:
Java is not coffee.
Java is NOT COFFEE.
The two lines are not equal.
The two lines are not equal.
But the lines are equal, ignoring case.
```

Display 3.3
Testing Strings for Equality

Lexicographic Order

- Lexicographic order is similar to alphabetical order, but is it based on the order of the characters in the ASCII (and Unicode) character set.
  - All the digits come before all the letters.
  - All the uppercase letters come before all the lower case letters.

Lexicographic Order, cont.

- Strings consisting of alphabetical characters can be compared using method compareTo and method toUpperCase or method toLowerCase.

```
String s1 = “Hello”;
String lowerS1 = s1.toLowerCase();
String s2 = “hello”;
if (lowerS1.compareTo(s2) == 0)
    System.out.println(“Equal!”);

//or use s1.compareToIgnoreCase(s2)
```

Method compareTo

- syntax

  String_1.compareTo(*String_2*)

- Method compareTo returns
  - a negative number if *String_1* precedes *String_2*
  - zero if the two strings are equal
  - a positive number of *String_2* precedes *String_1*
  - Tip: Think of compareTo is subtraction

Comparing Numbers vs. Comparing Strings

Integer and floating-point values	String objects
== !=	`equals( )` `equalsIgnoreCase( )`
>` < >= <=	`compareTo( )` `[lexicographical ordering]`

Short-Circuit Evaluation

- Java evaluates the && and || operators using a strategy called short-circuit mode in which it evaluates the right operand only if it needs to do so.
- For example, if n is 0, the right hand operand of && in

```
n != 0 && x % n == 0
```

is not evaluated at all because n != 0 is false. Because the expression

```
false && anything
```

is always false, the rest of the expression no longer matters.

- One of the advantages of short-circuit evaluation is that you can use && and || to prevent execution errors. If n were 0 in the earlier example, evaluating x % n would cause a "division by zero" error.

Designing for Change

- While it is clearly necessary for you to write programs that the compiler can understand, good programmers are equally concerned with writing code that *people* can understand.
- The importance of human readability arises from the fact that programs must be maintained over their life cycle.
- Typically, as much as 90 percent of the programming effort comes *after* the initial release of a system.
- There are several useful techniques that you can adopt to increase readability:
  - Use names that clearly express the purpose of variables and methods
  - Use proper indentation to make the structure of your programs clear
  - Use named constants to enhance both readability and maintainability

Precedence and associativity of Java operators.

- The table below shows all Java operators from highest to lowest precedence, along with their associativity.
- Most programmers do not memorize them all, and even those that do still use parentheses for clarity.

Precedence and associativity of Java operators.

Operator	Description	Level	Associativity
[ ] . ( ) ++ --	access array element access object member invoke a method post-increment post-decrement	1	left to right
++ -- + - ! ~	pre-increment pre-decrement unary plus unary minus logical NOT bitwise NOT	2	right to left
( ) new	cast object creation	3	right to left
* / %	multiplicative	4	left to right
+ - +	additive string concatenation	5	left to right
<< >> >>>	shift	6	left to right
< <= > >= instanceof	relational type comparison	7	left to right
== !=	equality	8	left to right
&	bitwise AND	9	left to right

|	bitwise OR	11	left to right
&&	conditional AND	12	left to right
||	conditional OR	13	left to right
?:	conditional	14	right to left
= += -= *= /= %= &= ^= |= <<= >>= >>>=	assignment	15	right to left

Flow of Control

- *Flow of control* is the order in which a program performs actions.
  - Up to this point, the order has been sequential.
- A *branching statement* chooses between two or more possible actions.
- A *loop statement* repeats an action until a stopping condition occurs.

Branching Statements: Outline

- The if-else Statement
- Introduction to Boolean Expressions
- Nested Statements and Compound Statements
- Multibranch if-else Statements
- The switch Statament
- (optional) The Conditional Operator

The if-else Statement

- A branching statement that chooses between two possible actions.
- syntax

```
if (Boolean_Expression)
    Statement_1
else
    Statement_2
```

The if-else Statement, cont.

- Example

```
if (count < 3)
    total = 0;
else
    total = total + count;
```

The if-else Statement, cont.

- class BankBalance

```
import java.util.*;
public class BankBalance
{
    public static final double OVERDRAWN_PENALTY = 8.00;
    public static final double INTEREST_RATE = 0.02;//2% annually
    public static void main(String[] args)
    {
        double balance;
        System.out.print("Enter your checking account balance: $");
        Scanner keyboard =  new Scanner(System.in);
        balance = keyboard.nextDouble();
        System.out.println("Original balance $" + balance);
        if (balance >= 0)
            balance = balance + (INTEREST_RATE * balance)/12;
        else
            balance = balance - OVERDRAWN_PENALTY;
        System.out.println("After adjusting for one month");
        System.out.println("of interest and penalties,");
        System.out.println("your new balance is $" + balance);
    }
}
```

Sample Screen Dialog 1

```
Enter your checking account balance: $505.67
Original balance $505.67
After adjusting for one month
of interest and penalties,
your new balance is $506.51278
```

Sample Screen Dialog 2

```
Enter your checking account balance: $-15.53
Original balance $-15.53
After adjusting for one month
of interest and penalties,
your new balance is $-23.53
```

Display 3.1
A Program Using if-else

Nested Statements

- An if-else statement can contain any sort of statement within it.
- In particular, it can contain another if-else statement.
  - An if-else may be nested within the “if” part.
  - An if-else may be nested within the “else” part.
  - An if-else may be nested within both parts.

Nested Statements, cont.

- syntax

```
if (Boolean_Expression_1)
    if (Boolean_Expression_2)
        Statement_1
    else
        Statement_2
else
    if (Boolean_Expression_3)
        Statement_3
    else
        Statement_4
```

Nested if Example

```
if (temperature > 90)  // int temperature
  if (sunny)           // boolean sunny
    System.out.println("Beach");
  else
    System.out.println("Movie");
else
  if (sunny)
    System.out.println("Tennis");
  else
    System.out.println("Volleyball");
```

Nested Statements, cont.

- Each else is paired with the **nearest unmatched** if.
- Indentation can communicate which if goes with which else.
- Braces are used to group statements.

Nested Statements, cont.

- Different indentation

<u>first form</u>

```
if (a > b)
  if (c > d)
        e = f;
else
  g = h;
```

<u>second form</u>

```
if (a > b)
  if (c > d)
            e = f;
else
            g = h;
```

Same to the compiler!

Nested Statements, cont.

- Are these different?

first form

```
if (a > b)
{
   if (c > d)
      e = f;
}
else
   g = h;
```

second form

```
if (a > b)
      if (c > d)
   e = f;
else
      g =h;
```

Nested Statements, cont.

- Proper indentation and nested if-else statements

"else" with outer "if"

```
if (a > b)
{
  if (c > d)
     e = f;
}
else
    g = h;
```

"else" with inner "if"

```
if (a > b)
   if (c > d)
      e = f;
   else
      g =h;
```

Compound Statements

- When a list of statements is enclosed in braces ({}), they form a single *compound statement.*
- syntax

```
{
    Statement_1;
     Statement_2;
   …
}
```

Compound Statements, cont.

- A compound statement can be used wherever a statement can be used.
- example

```
if (total > 10)
{
    sum = sum + total;
    total = 0;
}
```

Multibranch if-else Statements

- syntax

```
if (Boolean_Expression_1)
    Statement_1
else if (Boolean_Expression_2)
    Statement_2
else if (Boolean_Expression_3)
    Statement_3
else if …
else
    Default_Statement
```

Multibranch if-else Statements, cont.

- class Grader

```
import java.util.*;

public class Grader
{
    public static void main(String[] args)
    {
        int score;
        char grade;

        System.out.println("Enter your score: ");
        Scanner keyboard = new Scanner(System.in);
        score = keyboard.nextInt();

        if (score >= 90)
            grade = 'A';
        else if (score >= 80)
            grade = 'B';
        else if (score >= 70)
            grade = 'C';
        else if (score >= 60)
            grade = 'D';
        else
            grade = 'F';

        System.out.println("Score = " + score);
        System.out.println("Grade = " + grade);
    }
}
```

Sample Screen Dialog

```
Enter your score:
85
Score = 85
Grade = B
```

Display 3.4
Multibranch if-else Statement

Multibranch if-else Statements, cont.

- equivalent logically

```
if (score >= 90)
  grade = ‘A’;
if ((score >= 80) && (score < 90))
  grade = ‘B’;
if ((score >= 70) && (score < 80))
  grade = ‘C’;
if ((score >= 60) && (score < 70))
  grade = ‘D’;
if (score < 60)
  grade = ‘F’;
```

SWITCH STATEMENT

- The switch statement is a multiway branch that makes a decision based on an *integral* (integer or character) expression.
- The switch statement begins with the keyword switch followed by an integral expression in parentheses and called the *controlling expression.*

SWITCH STATEMENT, CONT.

- A list of cases follows, enclosed in braces.
- Each case consists of the keyword case followed by
  - a constant called the *case label*
  - a colon
  - a list of statements.
- The list is searched for a case label matching the controlling expression.

SWITCH STATEMENT, CONT.

- The action associated with a matching case label is executed.
- If no match is found, the case labeled default is executed.
  - The default case is optional, but recommended, even if it simply prints a message.
- Repeated case labels are not allowed.

SWITCH STATEMENT, CONT.

- `class MultipleBirths`

```
import java.util.*;

public class MultipleBirths
{
    public static void main(String[] args)
    {
        int numberOfBabies;
        System.out.print("Enter number of babies: ");
        Scanner keyboard = new Scanner(System.in);
        numberOfBabies = keyboard.nextInt();

        switch (numberOfBabies)
        {
            case 1:
                System.out.println("Congratulations.");
                break;
            case 2:
                System.out.println("Wow. Twins.");
                break;
            case 3:
                System.out.println("Wow. Triplets.");
                break;
            case 4:
            case 5:
                System.out.println("Unbelieveable.");
                System.out.println(numberOfBabies + " babies");
                break;
            default:
                System.out.println("I don't believe you.");
                break;
        }
    }
}
```

controlling expression

case label

break statement

Sample Screen Dialog 1

```
Enter number of babies: 1
Congratulations.
```

Sample Screen Dialog 2

```
Enter number of babies: 3
Wow. Triplets.
```

Sample Screen Dialog 3

```
Enter number of babies: 4
Unbelievable.
4 babies
```

Sample Screen Dialog 4

```
Enter number of babies: 6
I don't believe you.
```

Display 3.5
A switch Statement

SWITCH STATEMENT, CONT.

- The action for each case typically ends with the word break.
- The optional break statement prevents the consideration of other cases.
- The controlling expression can be anything that evaluates to an integral type (integer or character).

The switch Statement, cont.

- syntax

```
switch (Controlling_Expression)
{
    case Case_Label:
          Statement(s);
          break;
    case Case_Label:
    …
    default:
    …
}
```

Switch with char Type

```
char grade = 'A';
switch(grade)
{
   case 'A':
   case 'B':
   case 'C':
   case 'D':
            System.out.println("Pass");
            break;
   case 'W':
            System.out.println("Withdraw");
            break;
   case 'I':
            System.out.println("Incomplete");
            break;
   default:
            System.out.println("Fail");
}
```

Conditional Operator

```
if (n1 > n2)
        max = n1;
  else
        max = n2;
```

can be written as

```
max = (n1 > n2) ? n1 : n2;
```

- The ? and : together is called the *conditional operator* (a *ternary* operator).
- Note (n1 > n2) ? n1 : n2 is an *expression* that has a value unlike the "normal" if statement

Conditional Operator, cont.

- The conditional operator can be useful with print statements.

```
System.out.print("You worked " + hours + " " +
        ((hours > 1) ? "hours" : "hour"));
```

Summary of branching

- if statement (1 or 2 branches)
- Multi-branch if-else-if statement (3 or more branches)
- Multi-branch switch statement
- Conditional operator ? :

Loop Statements

- A portion of a program that repeats a statement or a group of statements is called a *loop*.
- The statement or group of statements to be repeated is called the *body* of the loop.
- A loop could be used to compute grades for each student in a class.
- There must be a means of exiting the loop.

Loop Structure

1. Control of loop: ***ICU***
   1. ***Initialization***
   2. ***Condition for termination (continuing)***
   3. ***Updating the condition***
2. Body of loop

Loop Statements

- the while Statement
- the do-while Statement
- the for Statement

WHILE STATEMENT

- also called a while loop
- a controlling boolean expression
  - True -> repeats the statements in the loop body
  - False -> stops the loop
  - Initially false (the very first time)
    - loop body will not even execute once

WHILE STATEMENT, CONT.

- syntax

while (*Boolean_Expression*)

    Body_Statement

or

while (*Boolean_Expression*)

{

    First_Statement

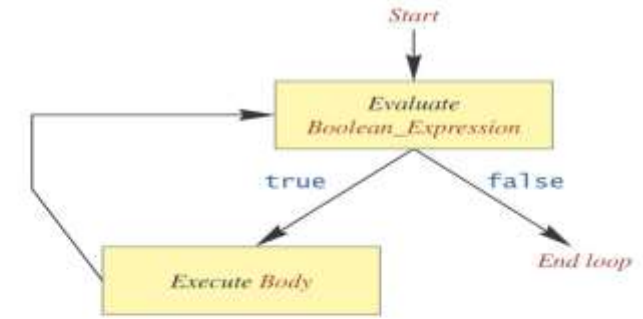
    Second_Statement

    ...

}

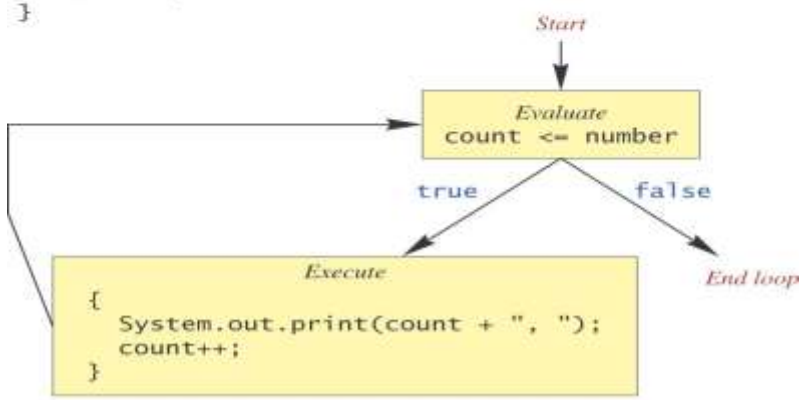
WHILE STATEMENT, CONT.

```
while (Boolean_Expression)
    Body
```



Example:

```
while (count <= number)
{
    System.out.print(count + ", ");
    count++;
}
```



Display 3.7
Semantics of the while Statement[2]

WHILE STATEMENT, CONT.

- `class WhileDemo`

```
import java.util.*;

public class WhileDemo
{
    public static void main(String[] args)
    {
        int count, number;

        System.out.println("Enter a number");
        Scanner keyboard = new Scanner(System.in);
        number = keyboard.nextInt();

        count = 1;
        while (count <= number)
        {
            System.out.print(count + ", ");
            count++;
        }

        System.out.println();
        System.out.println("Buckle my shoe.");
    }
}
```

Sample Screen Dialog 1

```
Enter a number:
2
1, 2,
Buckle my shoe.
```

Sample Screen Dialog 2

```
Enter a number:
3
1, 2, 3,
Buckle my shoe.
```

Sample Screen Dialog 3

```
Enter a number:
0

Buckle my shoe.
```

The loop body is iterated zero times.

Display 3.6
A while Loop

DO-WHILE STATEMENT

- also called a do-while loop (repeat-until loop)
- similar to a while statement
  - except that the loop body is executed at least once
- syntax

  do

  Body_Statement

  while (*Boolean_Expression*);

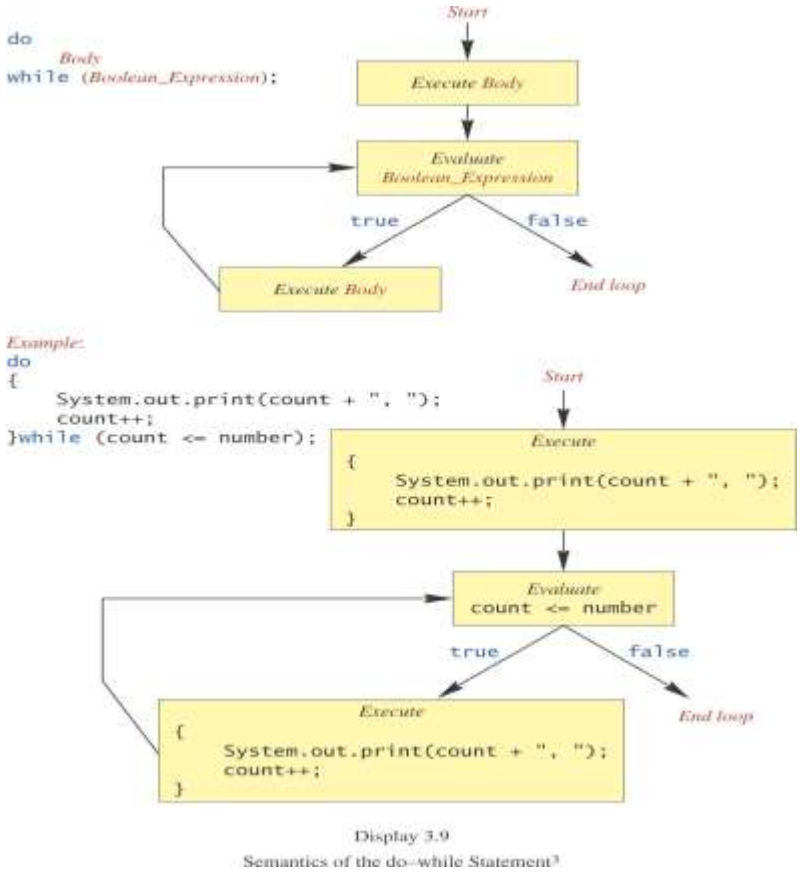
  - **don't forget the semicolon at the end!**

DO-WHILE STATEMENT, CONT.

- First, the loop body is executed.
- Then the boolean expression is checked.
  - As long as it is true, the loop is executed again.
  - If it is false, the loop exits.
- equivalent while statement

Statement(s)_S1
while (*Boolean_Condition*)
    Statement(s)_S1

DO-WHILE STATEMENT, CONT.



Display 3.9
Semantics of the do–while Statement³

DO-WHILE STATEMENT, CONT.

- `class DoWhileDemo`

```
import java.util.*;
public class DoWhileDemo
{
    public static void main(String[] args)
    {
        int count, number;
        System.out.println("Enter a number");
        Scanner keyboard = new Scanner(System.in);
        number = keyboard.nextInt();
        count = 1;
        do
        {
            System.out.print(count + ", ");
            count++;
        }while (count <= number);
        System.out.println();
        System.out.println("Buckle my shoe.");
    }
}
```

Sample Screen Dialog 1

```
Enter a number:
2
1, 2,
Buckle my shoe.
```

Sample Screen Dialog 2

```
Enter a number:
3
1, 2, 3,
Buckle my shoe.
```

Sample Screen Dialog 3

```
Enter a number:
0
1,
Buckle my shoe.
```

The loop body is always executed at least one time.

Display 3.8
A do–while Loop

Programming Example: Bug Infestation

- given
  - volume of a roach: 0.0002 cubic feet
  - starting roach population
  - rate of increase: 95%/week
  - volume of a house
- find
  - number of weeks to exceed the capacity of the house
  - number and volume of roaches

Programming Example: Bug Infestation, cont.

- `class BugTrouble`

```
import java.util.*;

/**
 Program to calculate how long it will take a population of
 roaches to completely fill a house from floor to ceiling.
*/
public class BugTrouble
{
    public static final double GROWTH_RATE = 0.95;//95% per week
    public static final double ONE_BUG_VOLUME = 0.002;//cubic feet

    public static void main(String[] args)
    {
        System.out.println("Enter the total volume of your house");
        System.out.print("in cubic feet: ");
        Scanner keyboard = new Scanner(System.in);
        double houseVolume = keyboard.nextDouble();

        System.out.println("Enter the estimated number of");
        System.out.print("roaches in your house: ");
        int startPopulation = keyboard.nextInt();
        int countWeeks = 0;
        double population = startPopulation;
        double totalBugVolume = population*ONE_BUG_VOLUME;

        while (totalBugVolume < houseVolume)
        {
            population = population + (GROWTH_RATE*population);
            totalBugVolume = population*ONE_BUG_VOLUME;
            countWeeks++;
        }

        System.out.println("Starting with a roach population of "
                                                + startPopulation);
        System.out.println("and a house with a volume of "
                                + houseVolume + " cubic feet,");
        System.out.println("after " + countWeeks + " weeks,");
        System.out.println("the house will be filled");
        System.out.println("floor to ceiling with roaches.");
        System.out.println("There will be " + (int)population + " roaches.");
        System.out.println("They will fill a volume of "
                            + (int)totalBugVolume + " cubic feet.");
        System.out.println("Better call Debugging Experts Inc.");
    }
}
```

(int) is a type cast as discussed in Chapter 2.

Sample Screen Dialog

```
Enter the total volume of your house
in cubic feet: 20000
Enter the estimated number of
roaches in your house: 100
Starting with a roach population of 100
and a house with a volume of 20000.0 cubic feet,
after 18 weeks,
the house will be filled
floor to ceiling with roaches.
There will be 16619693 roaches.
They will fill a volume of 33239 cubic feet.
Better call Debugging Experts Inc.
```

Display 3.10
Roach Population Program

Infinite Loops

- A loop which repeats without ever ending
- the controlling boolean expression (condition to continue)
  - never becomes false
- A negative growth rate in the preceding problem causes totalBugVolume always to be less than houseVolume
  - the loop never ends.

FOR STATEMENT

- A for statement executes the body of a loop a fixed number of times.
- example

```
for (count = 1; count < 3; count++)
        System.out.println(count);
System.out.println("Done");
```

FOR STATEMENT, CONT.

- syntax

  for (*Initialization*; *Condition*; *Update*)

  Body_Statement

  - *Body_Statement*
    - a simple statement or
    - a compound statement in {}.

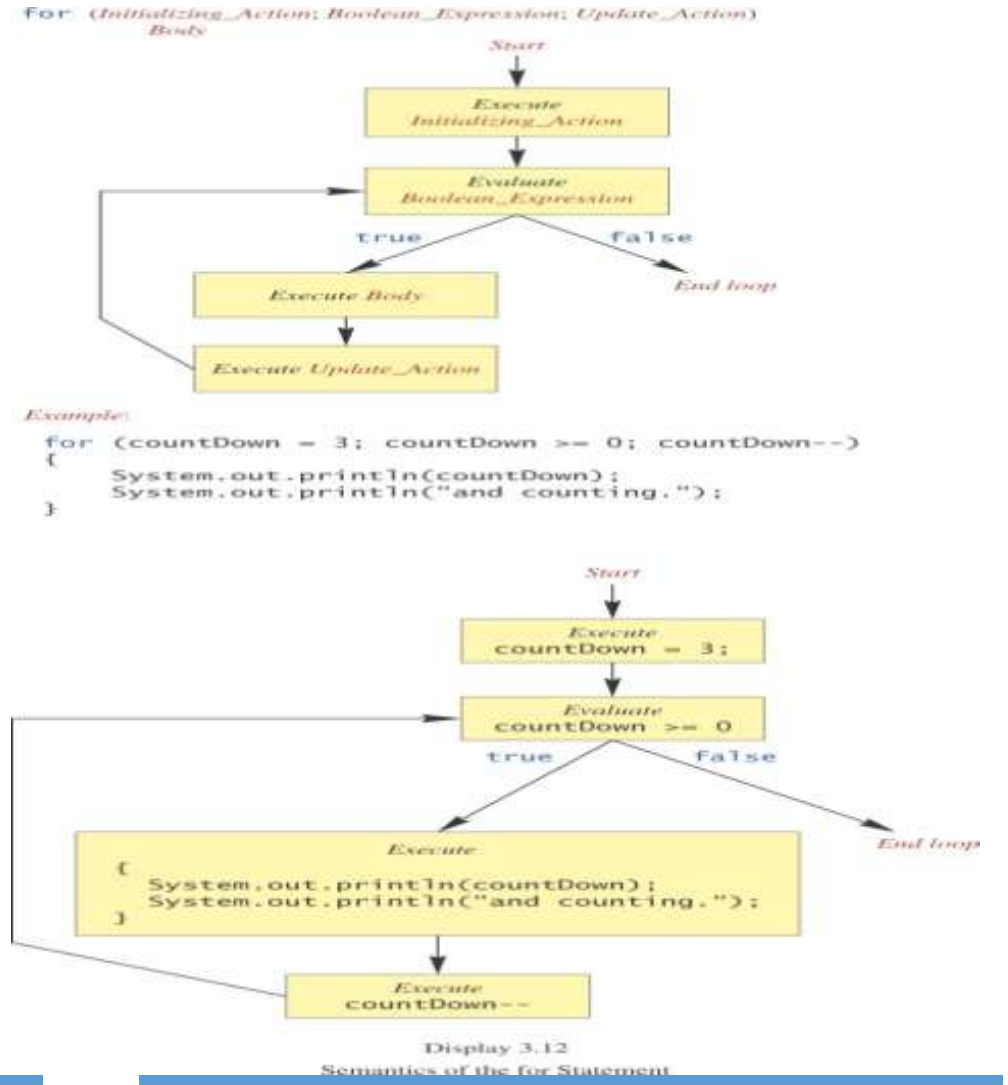
- corresponding while statement

  Initialization

  while *(Condition)*

  Body_Statement_Including_Update

for Statement, cont.



Display 3.12
Semantics of the for Statement

FOR STATEMENT, CONT.

- `class ForDemo`

```
public class ForDemo
{
    public static void main(String[] args)
    {
        int countDown;

        for (countDown = 3; countDown >= 0; countDown  )
        {
            System.out.println(countDown);
            System.out.println("and counting.");
        }

        System.out.println("Blast off!");
    }
}
```

Screen Output

```
3
and counting.
2
and counting.
1
and counting.
0
and counting.
Blast off!
```

Display 3.11
A for Statement

Multiple Initialization, etc.

- example

```
for (n = 1, p = 1; n < 10; n++)
    p = p * n
```

- Only one boolean expression is allowed, but it can consist of &&s, ||s, and !s.
- Multiple update actions are allowed, too.

```
for (n = 1, p = 100; n < p; n++, p -= n)
```

- rarely used

Choosing a Loop Statement

- If you know how many times the loop will be iterated, use a for loop.
- If you don't know how many times the loop will be iterated, but
  - it could be zero, use a while loop
  - it will be at least once, use a do-while loop.
- Generally, a while loop is a safe choice.

Summary of loop statements

- while loop
- do-while loop
- for loop

BREAK STATEMENT IN LOOPS: NOT RECOMMENDED

- A break statement can be used to end a loop immediately.
- The break statement ends only the **innermost** loop that contains the break statement.
- break statements make loops **more difficult** to understand:
  - Loop could end at different places (multiple possible exit points), harder to know where.
- Always try to end a loop at only one place--makes debugging easier (only one possible exit point)

Misuse of break Statements in loops

- “Because of the complications they introduce, break statements in loops should be avoided.
- Some authorities contend that a break statement should never be used to end a loop,
- but virtually all programming authorities agree that they should be used at most sparingly.”

EXIT METHOD

- Sometimes a situation arises that makes continuing the program pointless.
- A program can be terminated normally by

  System.exit(0).
- example

```
if (numberOfWinners == 0)
{
        System.out.println("/ by 0");
        System.exit(0);
}
```

Arrays in JAVA

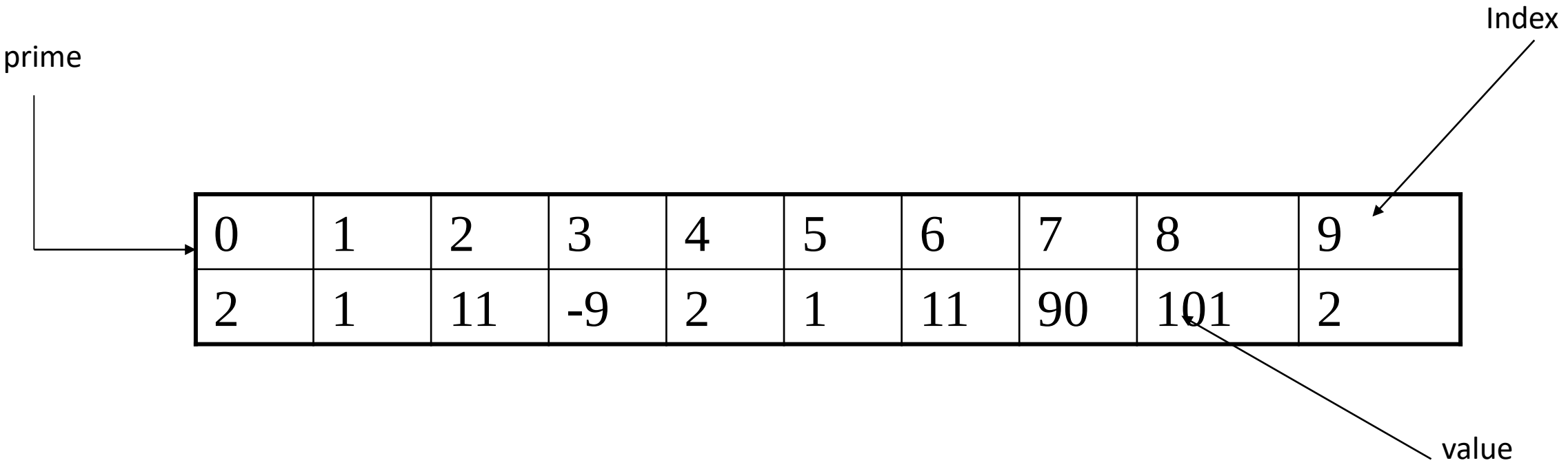
Declaring an Array Variable

- Do not have to create an array while declaring array variable
  - <*data type*> [] variable_name;
  - *int* [] prime;
  - *int* prime[];
- Both syntaxes are equivalent
- No memory allocation at this point

Defining an Array

- Define an array as follows:
  - variable_name=new <*data type*>[N];
  - primes=new int[10];
- Declaring and defining in the same statement:
  - *int*[] primes=new *int*[10];
- In JAVA, int is of 4 bytes, total space=4*10=40 bytes

Graphical Representation



0	1	2	3	4	5	6	7	8	9
2	1	11	-9	2	1	11	90	101	2

What happens if ...

- We define
  - int[] prime=new long[20];

```
MorePrimes.java:5: incompatible types
found: long[]
required: int[]
int[] primes = new long[20];
                    ^
```

- The right hand side defines an array, and thus the array variable should refer to the same type of array

Array Size through Input

```
BufferedReader stdin = new BufferedReader (new InputStreamReader(System.in));
String inData;
int    num;
System.out.println("Enter a Size for Array:");
inData = stdin.readLine();
num    = Integer.parseInt( inData ); // convert inData to int
long[] primes = new long[num];
System.out.println(“Array Length=”+primes.length);
….
```

SAMPLE RUN:

```
Enter a Size for Array:
4
Array Length=4
```

Default Initialization

- When array is created, array elements are initialized
  - Numeric values (int, double, etc.) to 0
  - Boolean values to false
  - Class types to null

Accessing Array Elements

- Index of an array is defined as
  - Positive int, byte or short values
  - Expression that results into these types
- Any other types used for index will give error
  - long, double, etc.
  - Incase Expression results in long, then type cast to int
- Indexing starts from 0 and ends at N-1

```
primes[2]=0;
int k = primes[2];
...
```

Validating Indexes

- JAVA checks whether the index values are valid at runtime
  - If index is negative or greater than the size of the array then an IndexOutOfBoundException will be thrown
  - Program will normally be terminated unless handled in the try {} catch {}

What happens if ...

```
long[] primes = new long[20];
primes[25]=33;
….
```

Runtime Error:

Exception in thread “main” java.lang.ArrayIndexOutOfBoundsException: 25

at MorePrimes.main(MorePrimes.java:6)

Reusing Array Variables

- Array variable is separate from array itself
  - Like a variable can refer to different values at different points in the program
  - Use array variables to access different arrays

    int[] primes=new int[10];

    ……

    primes=new int[50];
- Previous array will be discarded
- Cannot alter the type of array

Demonstration

```
long[] primes = new long[20];
primes[0] = 2;
primes[1] = 3;
long[] primes2=primes;
System.out.println(primes2[0]);
primes2[0]=5;
System.out.println(primes[0]);
```

OUTPUT

2

5

Array Length

- Refer to array length using *length*
  - A data member of array object
  - array_variable_name.length
  - for(int k=0; k<primes.length;k++)

    ….
- Sample Code:

```
long[] primes = new long[20];
System.out.println(primes.length);
```

- Output: 20

*Program taken from: http://chortle.ccsu.edu/CS151/Notes/chap47/ch47_10.html

Sample Program

```
class MinAlgorithm
{
   public static void main ( String[] args )
   {
         int[] array = { -20, 19, 1, 5, -1, 27, 19, 5 } ;
         int min=array[0]; // initialize the current minimum
         for ( int index=0; index < array.length; index++ )
            if ( array[ index ] < min )
                        min = array[ index ] ;
         System.out.println("The minimum of this array is: " + min );
   }
}
```

Arrays of Arrays

- Two-Dimensional arrays
  - float[][] temperature=new float[10][365];
  - 10 arrays each having 365 elements
  - First index: specifies array (row)
  - Second Index: specifies element in that array (column)
  - In JAVA float is 4 bytes, total Size=4*10*365=14,600 bytes

Initializing Array of Arrays

```
int[][] array2D = { {99, 42, 74, 83, 100}, {90, 91, 72, 88, 95}, {88,
  61, 74, 89, 96}, {61, 89, 82, 98, 93}, {93, 73, 75, 78, 99}, {50,
  65, 92, 87, 94}, {43, 98, 78, 56, 99} };
//5 arrays with 5 elements each
```

Initializing Varying Size Arrays

```
int[][] uneven = { { 1, 9, 4 }, { 0, 2}, { 0, 1, 2, 3, 4 } };
//Three arrays
//First array has 3 elements
//Second array has 2 elements
//Third array has 5 elements
```

INHERITANCE

- Inheritance is one of the cornerstones of object-oriented programming because it allows the creation of hierarchical classifications.
- In the terminology of Java, a class that is inherited is called a *superclass*.
- The class that does the inheriting is called a *subclass*. Therefore, a subclass is a specialized version of a superclass.

It inherits all of the members defined by the superclass and adds its own, unique elements.

Inheritance Basics

- To inherit a class, you simply incorporate the definition of one class into another by using the **extends** keyword. To see how, let's begin with a short example.
- The following program creates a superclass called **A** and a subclass called **B**.
- Notice how the keyword **extends** is used to create a subclass of **A**.

// A SIMPLE EXAMPLE OF INHERITANCE

```
// Create a superclass.
class A {
int i, j;
void showij() {
System.out.println("i and j: " + i + " " + j);
}
}
// Create a subclass by extending class A.
class B extends A {
int k;
void showk() {
System.out.println("k: " + k);
}
void sum() {
System.out.println("i+j+k: " + (i+j+k));
}
}
```

```
class SimpleInheritance {
public static void main(String args []) {
A superOb = new A();
B subOb = new B();
/ The superclass may be used by itself.
superOb.i = 10;
superOb.j = 20;
System.out.println("Contents of superOb: ");
superOb.showij();
System.out.println();
/* The subclass has access to all public members of
its superclass. */
subOb.i = 7;
subOb.j = 8;
subOb.k = 9;
System.out.println("Contents of subOb: ");
subOb.showij();
subOb.showk();
```

OUTPUT

- The output from this program is shown here:

Contents of superOb:

i and j: 10 20

Contents of subOb:

i and j: 7 8

k: 9

Sum of i, j and k in subOb:

i+j+k: 24

As you can see, the subclass **B** includes all of the members of its superclass, **A**. This is

why **subOb** can access **i** and **j** and call **showij( )**. Also, inside **sum( )**, **i** and **j** can be referred

to directly, as if they were part of **B**.

Even though **A** is a superclass for **B**, it is also a completely independent, stand-alone

class. Being a superclass for a subclass does not mean that the superclass cannot be used

by itself. Further, a subclass can be a superclass for another subclass.

- SYNTAX:

The general form of a **class** declaration that inherits a superclass is shown here:

class *subclass-name* extends *superclass-name*

{

// body of class

}

Member Access and Inheritance

- Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as **private**. For example, consider the following simple class hierarchy:
- In a class hierarchy, private members remain private to their class.
- This program will not compile because the use of **j** inside the **sum( )** method of **B** causes an access violation. Since **j** is declared as **private**, it is only accessible by other members of its own class. Subclasses have no access to it.

EXAMPLE

```
// Create a superclass.

class A {
int i; // public by default
private int j; // private to A
void setij(int x, int y) {
i = x;
j = y;
}
}

// A's j is not accessible here.
class B extends A {
int total;
void sum() {
total = i + j; // ERROR, j is not accessible here
}
}
```

```
class Access
{
public static void main(String args[])
 {
B subOb = new B();
subOb.setij(10, 12);
subOb.sum();
System.out.println("Total is " + subOb.total);

}
}
```

A Superclass Variable Can Reference a Subclass Object

- A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass.

```
class RefDemo
{
public static void main(String args[]) {
BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
Box plainbox = new Box();
double vol;
vol = weightbox.volume();
System.out.println("Volume of weightbox is " + vol);
System.out.println("Weight of weightbox is " +
weightbox.weight);
System.out.println();
// assign BoxWeight reference to Box reference
plainbox = weightbox;
vol = plainbox.volume(); // OK, volume() defined in Box
System.out.println("Volume of plainbox is " + vol);
/* The following statement is invalid because plainbox
does not define a weight member. */
// System.out.println("Weight of plainbox is " + plainbox.weight);
}
}
```

A Superclass Variable Can Reference a Subclass Object

- **weightbox** is a reference to **BoxWeight** objects, and **plainbox** is a reference to **Box** objects.
- Since **BoxWeight** is a subclass of **Box**, it is permissible to assign **plainbox** a reference to the **weightbox** object.
- It is important to understand that it is the type of the reference variable—not the type of the object that it refers to—that determines what members can be accessed.
- That is, when a reference to a subclass object is assigned to a superclass reference variable, you will have access only to those parts of the object defined by the superclass.
- This is why **plainbox** can't access **weight** even when it refers to a **BoxWeight** object. If you think about it, this makes sense, because the superclass has no knowledge of what a subclass adds to it. This is why the last line of code in the preceding fragment is commented out. It is not possible for a **Box** reference to access the **weight** field, because **Box** does not define one.

USING SUPER

- Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super**.
- **super** has two general forms.
- The first calls the superclass' constructor.
- The second is used to access a member of the superclass that has been hidden by a member of a subclass.

Using super to Call Superclass Constructors

- A subclass can call a constructor defined by its superclass by use of the following form of
- **super**:

super(*arg-list*);

- Here, *arg-list* specifies any arguments needed by the constructor in the superclass. **super( )** must always be the first statement executed inside a subclass' constructor.
- To see how **super( )** is used, consider this improved version of the **BoxWeight** class:

SUPER CLASS CONSTRUCTOR

```
// BoxWeight now uses super to initialize its Box attributes.
class BoxWeight extends Box {
double weight; // weight of box
// initialize width, height, and depth using super()
BoxWeight(double w, double h, double d, double m) {
super(w, h, d); // call superclass constructor
weight = m;
}
}
```

- **BoxWeight( )** calls **super( )** with the arguments **w**, **h**, and **d**. This causes the **Box** constructor to be called, which initializes **width**, **height**, and **depth** using these values.
- **BoxWeight** no longer initializes these values itself. It only needs to initialize the value unique to it: **weight**. This leaves **Box** free to make these values **private** if desired.

```
// A complete implementation of BoxWeight.
class Box {
private double width;
private double height;
private double depth;
// construct clone of an object
Box(Box ob) { // pass object to constructor
width = ob.width;
height = ob.height;
depth = ob.depth;
}
// constructor used when all dimensions specified
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// constructor used when no dimensions specified
Box() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}
// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
class BoxWeight extends Box {
double weight; // weight of box
// construct clone of an object
BoxWeight(BoxWeight ob) { // pass object to constructor
super(ob);
weight = ob.weight;
}
// constructor when all parameters are specified
BoxWeight(double w, double h, double d, double m) {
super(w, h, d); // call superclass constructor
weight = m;
}
// default constructor
BoxWeight() {
super();
weight = -1;
}
// constructor used when cube is created
BoxWeight(double len, double m) {
super(len);
weight = m;
}
}
```

```
class DemoSuper {
public static void main(String args[]) {
BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
BoxWeight mybox3 = new BoxWeight(); // default
BoxWeight mycube = new BoxWeight(3, 2);
BoxWeight myclone = new BoxWeight(mybox1);
double vol;
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
System.out.println("Weight of mybox1 is " + mybox1.weight);
System.out.println();
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
System.out.println("Weight of mybox2 is " + mybox2.weight);
System.out.println();
vol = mybox3.volume();
System.out.println("Volume of mybox3 is " + vol);
System.out.println("Weight of mybox3 is " + mybox3.weight);
System.out.println();
vol = myclone.volume();
System.out.println("Volume of myclone is " + vol);
System.out.println("Weight of myclone is " + myclone.weight);
System.out.println();
vol = mycube.volume();
System.out.println("Volume of mycube is " + vol);
System.out.println("Weight of mycube is " + mycube.weight);
System.out.println();
}
}
```

This program generates the following output:

```
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3
Volume of mybox2 is 24.0
Weight of mybox2 is 0.076
Volume of mybox3 is -1.0
Weight of mybox3 is -1.0
Volume of myclone is 3000.0
Weight of myclone is 34.3
Volume of mycube is 27.0
Weight of mycube is 2.0
```

Pay special attention to this constructor in BoxWeight:

```
// construct clone of an object
BoxWeight(BoxWeight ob) { // pass object to constructor
super(ob);
weight = ob.weight;
```

SUPER

- Notice that **super( )** is passed an object of type **BoxWeight**—not of type **Box**. This still invokes the constructor **Box(Box ob)**.
- As mentioned earlier, a superclass variable can be used to reference any object derived from that class. Thus, we are able to pass a **BoxWeight** object to the **Box** constructor. Of course, **Box** only has knowledge of its own members.

A Second Use for super

- The second form of **super** acts somewhat like **this**, except that it always refers to the superclass of the subclass in which it is used.
- This usage has the following general form:

  super.*member*

- Here, *member* can be either a method or an instance variable.
- This second form of **super** is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.
- Consider this simple class hierarchy:

// Using super to overcome name hiding.

```
class A {
int i;
}
// Create a subclass by extending class A.
class B extends A {
int i; // this i hides the i in A
B(int a, int b) {
super.i = a; // i in A
i = b; // i in B
}
```

```
void show() {
System.out.println("i in superclass: " + super.i);
System.out.println("i in subclass: " + i);
}
}
class UseSuper {
public static void main(String args[]) {
B subOb = new B(1, 2);
subOb.show();
}
}
```

This program displays the following:

```
i in superclass: 1
i in subclass: 2
```

Although the instance variable **i** in **B** hides the **i** in **A**, **super** allows access to the **i** defined in the superclass. As you will see, **super** can also be used to call methods that are hidden by a subclass.

Creating a Multilevel Hierarchy

- given three classes called **A**, **B**, and **C**, **C** can be a subclass of **B**, Which is a subclass of **A**. When this type of situation occurs, each subclass inherits all of the traits found in all of its superclasses. In this case, **C** inherits all aspects of **B** and **A**.

When Constructors Are Executed

- When a class hierarchy is created, in what order are the constructors for the classes that make up the hierarchy executed? For example, given a subclass called **B** and a superclass called **A**, is **A**'s constructor executed before **B**'s, or vice versa?
- answer is that in a class hierarchy, constructors complete their execution in order of derivation, from superclass to

  subclass.

Further, since **super( )** must be the first statement executed in a subclass constructor, this order is the same whether or not **super( )** is used.

If **super( )** is not used, then the default or parameterless constructor of each superclass will be executed

```
// Demonstrate when constructors are executed.
// Create a super class.
class A {
A() {
System.out.println("Inside A's constructor.");
}
}
// Create a subclass by extending class A.
class B extends A {
B() {
System.out.println("Inside B's constructor.");
}
}
// Create another subclass by extending B.
class C extends B {
C() {
System.out.println("Inside C's constructor.");
}
}
class CallingCons {
public static void main(String args[]) {
C c = new C();
}
}
```

OUTPUT

- The output from this program is shown here:
- Inside A's constructor
- Inside B's constructor
- Inside C's constructor
- As you can see, the constructors are executed in order of derivation.
- If you think about it, it makes sense that constructors complete their execution in order of derivation. Because a superclass has no knowledge of any subclass, any initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by the subclass. Therefore, it must complete its execution first.

Method Overriding

- In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass.
- When an overridden method is called from within its subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden

```
// Method overriding.
class A {
int i, j;
A(int a, int b) {
i = a;
j = b;
}
// display i and j
void show() {
System.out.println("i and j: " + i +" " + j);
}
}
class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
}
// display k – this overrides show() in A
void show() {
System.out.println("k: " + k);
}
}
class Override {
public static void main(String args[]) {
B subOb = new B(1, 2, 3);
subOb.show(); // this calls show() in B
}
}
```

The output produced by this program is shown here:

```
k: 3
```

- When **show( )** is invoked on an object of type **B**, the version of **show( )** defined within **B** is used. That is, the version of **show( )** inside **B** overrides the version declared in **A**.
- If you wish to access the superclass version of an overridden method, you can do so by using **super**. For example, in this version of **B**, the superclass version of **show( )** is invoked within the subclass' version. This allows all instance variables to be displayed.

```
class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
}
void show() {
super.show(); // this calls A's show()
System.out.println("k: " + k);
}
}
```

If you substitute this version of **A** into the previous program, you will see the following output:

```
i and j: 1 2
k: 3
```

Here, **super.show( )** calls the superclass version of **show( )**.

- Method overriding occurs *only* when the names and the type signatures of the two methods are identical. If they are not, then the two methods are simply overloaded. For
- example, consider this modified version of the preceding example:

```
// Methods with differing type signatures are overloaded – not
// overridden.
class A {
int i, j;
A(int a, int b) {
i = a;
j = b;
}
// display i and j
void show() {
System.out.println("i and j: " + i + " " + j);
}
}
// Create a subclass by extending class A.
class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
}
// overload show()
void show(String msg) {
System.out.println(msg + k);
}
}
```

```
Override {
public static void main(String args[]) {
B subOb = new B(1, 2, 3);
subOb.show("This is k: "); // this calls show() in B
subOb.show(); // this calls show() in A
}
}
```

The output produced by this program is shown here:

```
This is k: 3
i and j: 1 2
```

The version of **show( )** in **B** takes a string parameter. This makes its type signature

different from the one in **A**, which takes no parameters. Therefore, no overriding (or name

hiding) takes place. Instead, the version of **show( )** in **B** simply overloads the version of

show( ) in **A**.

Dynamic Method Dispatch

- Method overriding forms the basis for one of Java's most powerful concepts: *dynamic method dispatch*.
- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
- Dynamic method dispatch is important because this is how Java implement run-time polymorphism.

- When different types of objects are referred to, different versions of an overridden method will be called. In other words, *it is the type of the object being referred to* (not the type of the reference variable) that determines which version of an overridden method will be executed.
- Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

Here is an example that illustrates dynamic method dispatch:

```
// Dynamic Method Dispatch
class A {
void callme() {
System.out.println("Inside A's callme method");
}
}
class B extends A {
// override callme()
void callme() {
System.out.println("Inside B's callme method");
}
}
class C extends A {
// override callme()
void callme() {
System.out.println("Inside C's callme method");
}
}
class Dispatch {
public static void main(String args[]) {
A a = new A(); // object of type A
B b = new B(); // object of type B
C c = new C(); // object of type C
A r; // obtain a reference of type A
r = a; // r refers to an A object
r.callme(); // calls A's version of callme
r = b; // r refers to a B object
r.callme(); // calls B's version of callme
r = c; // r refers to a C object
r.callme(); // calls C's version of callme
}
}
```

The output from the program is shown here:

```
Inside A's callme method
Inside B's callme method
Inside C's callme method
```

This program creates one superclass called **A** and two subclasses of it, called **B** and **C**. Subclasses **B** and **C** override **callme( )** declared in **A**. Inside the **main( )** method, objects of type **A**, **B**, and **C** are declared. Also, a reference of type **A**, called **r**, is declared. The program then in turn assigns a reference to each type of object to **r** and uses that reference to invoke

callme( ). As the output shows, the version of **callme( )** executed is determined by the type

of object being referred to at the time of the call. Had it been determined by the type of the reference variable, **r**, you would see three calls to **A**'s **callme( )** method.

Applying Method Overriding

- **Applying Method Overriding**
- Let's look at a more practical example that uses method overriding. The following program creates a superclass called **Figure** that stores the dimensions of a two-dimensional object.
- It also defines a method called **area( )** that computes the area of an object.
- The program derives two subclasses from **Figure**. The first is **Rectangle** and the second is **Triangle**.
- Each of these subclasses overrides **area( )** so that it returns the area of a rectangle and a triangle, respectively.

```
// Using run-time polymorphism.
class Figure {
double dim1;
double dim2;
Figure(double a, double b) {
dim1 = a;
dim2 = b;
}
double area() {
System.out.println("Area for Figure is undefined.");
return 0;
}
}
class Rectangle extends Figure {
Rectangle(double a, double b) {
super(a, b);
}
// override area for rectangle
double area() {
System.out.println("Inside Area for Rectangle.");
return dim1 * dim2;
}
}
class Triangle extends Figure {
Triangle(double a, double b) {
super(a, b);
}
// override area for right triangle
double area() {
System.out.println("Inside Area for Triangle.");
return dim1 * dim2 / 2;
}
}
class FindAreas {
public static void main(String args[]) {
Figure f = new Figure(10, 10);
Rectangle r = new Rectangle(9, 5);
Triangle t = new Triangle(10, 8);
Figure figref;
figref = r;
System.out.println("Area is " + figref.area());
figref = t;
System.out.println("Area is " + figref.area());
figref = f;
System.out.println("Area is " + figref.area());
}
```

OUTPUT

- The output from the program is shown here:

Inside Area for Rectangle.

Area is 45

Inside Area for Triangle.

Area is 40

Area for Figure is undefined.

Area is 0

Through the dual mechanisms of inheritance and run-time polymorphism, it is possible to define one consistent interface that is used by several different, yet related, types of objects. In this case, if an object is derived from **Figure**, then its area can be obtained by calling **area( )**. The interface to this operation is the same no matter what type of figure is being used.

Using Abstract Classes

- You may have methods that must be overridden by the subclass in order for the subclass to have any meaning. Consider the class **Triangle**.
- It has no meaning if **area( )** is not defined. In this case, you want some way to ensure that a subclass does, indeed, override all necessary methods. Java's solution to this problem is the

  abstract method.
- You can require that certain methods be overridden by subclasses by specifying the **abstract** type modifier. These methods are sometimes referred to as *subclasser responsibility* because they have no implementation specified in the superclass.
- Thus, a subclass mustoverride them—it cannot simply use the version defined in the superclass.
- To declare an abstract method, use this general form:
- abstract *type name*(*parameter-list*);

- Any class that contains one or more abstract methods must also be declared abstract.
- To declare a class abstract, you simply use the **abstract** keyword in front of the **class** keyword at the beginning of the class declaration. There can be no objects of an abstract class.
- That is, an abstract class cannot be directly instantiated with the **new** operator. Such objects would be useless, because an abstract class is not fully defined.
- Also, you cannot declare abstract constructors, or abstract static methods.

```
// A Simple demonstration of abstract.
abstract class A {
abstract void callme();
// concrete methods are still allowed in abstract classes
void callmetoo() {
System.out.println("This is a concrete method.");
}
}
class B extends A {
void callme() {
System.out.println("B's implementation of callme.");
}
}
class AbstractDemo {
public static void main(String args[]) {
B b = new B();
b.callme();
b.callmetoo();
}
}
```

- no objects of class **A** are declared in the program. As mentioned, it is not possible to instantiate an abstract class. One other point: class **A** implements a concrete method called **callmetoo( )**. This is perfectly acceptable. Abstract classes can include as much implementation as they see fit.
- Although abstract classes cannot be used to instantiate objects, they can be used to create object references, because Java's approach to run-time polymorphism is implemented through the use of superclass references.
- Thus, it must be possible to create a reference to an abstract class so that it can be used to point to a subclass object.
- You will see this feature put to use in the next example

```
// Using abstract methods and classes.
abstract class Figure {
double dim1;
double dim2;
Figure(double a, double b) {
dim1 = a;
dim2 = b;
}
// area is now an abstract method
abstract double area();
}
class Rectangle extends Figure {
Rectangle(double a, double b) {
super(a, b);
}
// override area for rectangle
double area() {
System.out.println("Inside Area for Rectangle.");
return dim1 * dim2;
}
}
class Triangle extends Figure {
Triangle(double a, double b) {
super(a, b);
}
// override area for right triangle
double area() {
System.out.println("Inside Area for Triangle.");
return dim1 * dim2 / 2;
}
}
class AbstractAreas {
public static void main(String args[]) {
// Figure f = new Figure(10, 10); // illegal now
Rectangle r = new Rectangle(9, 5);
Triangle t = new Triangle(10, 8);
Figure figref; // this is OK, no object is created
figref = r;
System.out.println("Area is " + figref.area());
System.out.println("Area is " + figref.area());
}
}
```

- As the comment inside **main( )** indicates, it is no longer possible to declare objects of type **Figure**, since it is now abstract. And, all subclasses of **Figure** must override **area( )**.
- Toprove this to yourself, try creating a subclass that does not override **area( )**. You will receive a compile-time error.
- Although it is not possible to create an object of type **Figure**, you can create a reference variable of type **Figure**. The variable **figref** is declared as a reference to **Figure**, which means that it can be used to refer to an object of any class derived from **Figure**.
- As explained, it is through superclass reference variables that overridden methods are resolved at run time

Using final with Inheritance

- The keyword **final** has three uses.
- First, it can be used to create the equivalent of a namedconstant.
- The other two uses of **final** apply to inheritance.

Using final to Prevent Overriding

To disallow a method from being overridden,specify **final** as a modifier at the start of its declaration. Methods declared as **final** cannot be overridden. The following fragment illustrates **final**:

```
class A {
final void meth() {
System.out.println("This is a final method.");
}
}
class B extends A {
void meth() { // ERROR! Can't override.
System.out.println("Illegal!");
}
}
```

Because **meth( )** is declared as **final**, it cannot be overridden in **B**. If you attempt to do so, a compile-time error will result.

LATE BINDING & EARLY BINDING

- Methods declared as **final** can sometimes provide a performance enhancement: The compiler is free to *inline* calls to them because it "knows" they will not be overridden by a subclass. When a small **final** method is called, often the Java compiler can copy the bytecodefor the subroutine directly inline with the compiled code of the calling method,
- thus eliminating the costly overhead associated with a method call. Inlining is an option only with **final** methods.
- Normally, Java resolves calls to methods dynamically, at run time. This is called *late binding*.
- However, since **final** methods cannot be overridden, a call to one can be resolved at compile time. This is called early binding.

Using final to Prevent Inheritance

- Sometimes you will want to prevent a class from being inherited.
- To do this, precede the class declaration with **final**. Declaring a class as **final** implicitly declares all of its methods as **final**, too. As you might expect, it is illegal to declare a class as both **abstract** and **final** since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.
- Here is an example of a **final** class:

```
final class A {
//...
}
// The following class is illegal.
class B extends A { // ERROR! Can't subclass A
//...
}
```

The Object Class

- There is one special class, **Object**, defined by Java. All other classes are subclasses of **Object**.That is, **Object** is a superclass of all other classes. This means that a reference variable of type **Object** can refer to an object of any other class. Also, since arrays are implemented as classes, a variable of type **Object** can also refer to any array.
- **Object** defines the following methods, which means that they are available in every object.
- **Method Purpose**

Object clone( )---------- Creates a new object that is the same as the object being cloned.

boolean equals(Object *object*)---------- Determines whether one object is equal to another.

void finalize( ) ----------Called before an unused object is recycled.

Class<?> getClass( ) ----------Obtains the class of an object at run time.

int hashCode( ) ----------Returns the hash code associated with the invoking object.

void notify( )---------- Resumes execution of a thread waiting on the invoking object.

void notifyAll( ) Resumes execution of all threads waiting on the invoking object.

String toString( ) Returns a string that describes the object.

void wait( )

void wait(long *milliseconds*)

void wait(long *milliseconds*, int *nanoseconds*)

----------Waits on another thread of execution.

- The methods **getClass( )**, **notify( )**, **notifyAll( )**, and **wait( )** are declared as **final**. You may
- override the others
- However, notice two methods now: **equals( )** and **toString( )**. The **equals( )** method compares two objects. It returns **true** if the objects are equal, and **false** otherwise. The precise definition of equality can vary, depending on the type of objects being compared.
- The **toString( )** method returns a string that contains a description of the object on which it is called. Also, this method is automatically called when an object is output using **println( )**.
- Many classes override this method. Doing so allows them to tailor a description specifically for the types of objects that they create.
- One last point: Notice the unusual syntax in the return type for **getClass( )**. This relates to Java's *generics* feature,

PACKAGES

- *Packages* are containers for classes. They are used to keep the class name space compartmentalized.
- For example, a package allows you to create a class named **List**, which you can store in your own package without concern that it will collide with some other class named **List** stored elsewhere.

INTERFACES

- Java allows you to fully abstract an interface from its implementation.
- Using **interface**, you can specify a set of methods that can be implemented by one or more classes.

  In its traditional form, the **interface**, itself, does not actually define any implementation.

Although they are similar to abstract classes, **interface**s have an additional capability:

A class can implement more than one interface. By contrast, a class can only inherit a single superclass (abstract or otherwise).

Defining a Package

- To create a package is quite easy: simply include a **package** command as the first statement in a Java source file.
- Any classes declared within that file will belong to the specified package.
- The **package** statement defines a name space in which classes are stored.
- If you omit the **package** statement, the class names are put into the default package, which has no name.

- This is the general form of the **package** statement:
- package *pkg*;
- Here, *pkg* is the name of the package. For example, the following statement creates a package called **MyPackage**:
- package MyPackage;
- Java uses file system directories to store packages. For example, the **.class** files for any
- classes you declare to be part of **MyPackage** must be stored in a directory called **MyPackage**.

- You can create a hierarchy of packages.

The general form of a multileveled package

statement is shown here:

package *pkg1*[.*pkg2*[.*pkg3*]];

A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as

- package java.awt.image;
- needs to be stored in **java\awt\image** in a Windows environment

Finding Packages and CLASSPATH

- the Java run-time system uses the current working directory as its starting point.
- Thus, if your package is in a subdirectory of the current directory, it will be found.
- Second, you can specify a directory path or paths by setting the **CLASSPATH** environmental variable.
- Third, you can use the **-classpath** option with **java** and **javac** to specify the path to your classes.

- For example, consider the following package specification:
- package MyPack
- In order for a program to find **MyPack**, one of three things must be true.
- Either the program can be executed from a directory immediately above **MyPack**, or the **CLASSPATH** must be set to include the path to **MyPack**, or the **-classpath** option must specify the path to **MyPack** when the program is run via **java**

- When the second two options are used, the class path *must not* include **MyPack**, itself.
- It must simply specify the *path to* **MyPack**. For example, in a Windows environment, if the path

to **MyPack** is

C:\MyPrograms\Java\MyPack

then the class path to **MyPack** is

C:\MyPrograms\Java

```
// A simple package
package MyPack;
class Balance {
String name;
double bal;
Balance(String n, double b) {
name = n;
bal = b;
}
void show() {
if(bal<0)
System.out.print("--> ");
System.out.println(name + ": $" + bal);
}
}
```

```
class AccountBalance
 {
public static void main(String args[])
{
  Balance current[] = new Balance[3];
current[0] = new Balance("K. J. Fielding", 123.23);
current[1] = new Balance("Will Tell", 157.02);
current[2] = new Balance("Tom Jackson", -12.33);
for(int i=0; i<3; i++) current[i].show();
}
}
```

- Call this file **AccountBalance.java** and put it in a directory called **MyPack**. Next, compile the file. Make sure that the resulting **.class** file is also in the **MyPack** directory.
- Then, try executing the **AccountBalance** class, using the following command line:

- **java MyPack.AccountBalance**

- Remember, you will need to be in the directory above **MyPack** when you execute this command. (Alternatively, you can use one of the other two options described in thepreceding section to specify the path **MyPack**.)
- As explained, **AccountBalance** is now part of the package **MyPack**. This means that it cannot be executed by itself. That is, you cannot use this command line:
- **java AccountBalance**

AccountBalance must be qualified with its package name.

- Java provides many levels of protection to allow

fine-grained control over the visibility of variables and methods within classes, subclasses, and packages.

- Java
- addresses four categories of visibility for class members:

- • Subclasses in the same package
- • Non-subclasses in the same package
- • Subclasses in different packages
- • Classes that are neither in the same package nor subclasses

- The three access modifiers, **private**, **public**, and **protected**, provide a variety of ways to produce the many levels of access required by these categories. Table 9-1 sums up the interactions.

Class member access

- private ,no modifier, protected,public
- Same class Yes Yes Yes Yes
- Same package subclass No Yes Yes Yes
- Same package non-subclass No Yes Yes Yes
- Different package subclass No No Yes Yes
- Different package non-subclass No No No yes

An Access Example

- The source for the first package defines three classes: Protection, Derived, and SamePackage. The first class defines four int variables in each of the legal protection modes. The variable n is declared with the default protection, n_pri is private, n_pro is protected, and n_pub is public.
- Each subsequent class in this example will try to access the variables in an instance of this class. The lines that will not compile due to access restrictions are commented out. Before each of these lines is a comment listing the places from which this level of protection would allow access.
- The second class, Derived, is a subclass of Protection in the same package, p1. This grants Derived access to every variable in Protection except for n_pri, the private one.
- The third class, SamePackage, is not a subclass of Protection, but is in the same package and also has access to all but n_pri.

This is file Protection.java:

```
package p1;
public class Protection
 {
int n = 1;
private int n_pri = 2;
 protected int n_pro = 3;
public int n_pub = 4;
public Protection()
{
System.out.println("base constructor");
 System.out.println("n = " + n);
 System.out.println("n_pri = " + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
 }
}
```

This is file Derived.java:

```
package p1;
class Derived extends Protection
 {
Derived()
{
System.out.println("derived constructor"); System.out.println("n = " + n);
System.out.println("n_pri = "4 + n_pri); // class only // System.out.println("n_pro = " + n_pro); System.out.println("n_pub = " + n_pub);
 }
}
```

This is file SamePackage.java

```
 package p1;
class SamePackage
 {
SamePackage()
{
Protection p = new Protection(); System.out.println("same package constructor");
System.out.println("n = " + p.n);
// class only // System.out.println("n_pri = " + p.n_pri); System.out.println("n_pro = " +
p.n_pro); System.out.println("n_pub = " + p.n_pub);
}
}
```

This is file Protection2.java:

```
package p2;
class Protection2 extends p1.Protection
 {
Protection2()
 {
System.out.println("derived other package constructor");
// class or package only // System.out.println("n = " + n);
// class only // System.out.println("n_pri = " + n_pri); System.out.println("n_pro = " +
n_pro); System.out.println("n_pub = " + n_pub);
}
}
```

This is file OtherPackage.java:

```
package p2;
 class OtherPackage
{
OtherPackage()
 {
 p1.Protection p = new p1.Protection(); System.out.println("other packageconstructor"); // class or package only
 // System.out.println("n = " + p.n);
// class only
 // System.out.println("n_pri = " + p.n_pri);
 // class, subclass or package only
 // System.out.println("n_pro = " + p.n_pro);
 System.out.println("n_pub = " + p.n_pub);
 }
}
```

THE ONE FOR PACKAGE P1 IS SHOWN HERE:

If you want to try these two packages, here are two test files you can use.

```
// Demo package p1.
 package p1;
// Instantiate the various classes in p1.
public class Demo
{
public static void main(String args[])
 {
 Protection ob1 = new Protection();
 Derived ob2 = new Derived();
SamePackage ob3 = new SamePackage();
}
}
```

The test file for p2 is shown next:

```
// Demo package p2.
package p2;
// Instantiate the various classes in p2.
 public class Demo
 {
public static void main(String args[])
 {
Protection2 ob1 = new Protection2(); OtherPackage ob2 =new OtherPackage();
 }
}
```

Importing Packages

- Java includes the import statement to bring certain classes, or entire packages, into visibility. Once imported, a class can be referred to directly, using only its name.
- The import statement is a convenience to the programmer and is not technically needed to write a complete Java program.
- If you are going to refer to a few dozen classes in your application, however, the import statement will save a lot of typing.
- In a Java source file, import statements occur immediately following the package statement (if it exists) and before any class definitions.
- This is the general form of the import statement:
- import pkg1 [.pkg2].(classname | *);
- Here, pkg1 is the name of a top-level package, and pkg2 is the name of a subordinate package inside the outer package separated by a dot (.).

Package

- All of the standard Java classes included with Java are stored in a package called java.
- The basic language functions are stored in a package inside of the java package called java.lang.
- import java.lang.*;
- Finally, you specify either an explicit classname or a star (*), which indicates that the Java compiler should import the entire package.

It must be emphasized that the import statement is optional.

Any place you use a class name, you can use its fully qualified name, which includes its full package hierarchy.

For example, this fragment uses an import statement:

```
import java.util.*;
class MyDate extends Date { }
```

The same example without the import statement looks like this:

- class MyDate extends java.util.Date { }

```
package MyPack;
/* Now, the Balance class, its constructor, and its show() method are public. This means
that they can be used by non-subclass code outside their package. */
public class Balance
 {
String name;
 double bal;
 public Balance(String n, double b)
{
name = n; bal = b;
}
public void show()
 {
if(bal <0)
System.out.print("--> ");
System.out.println(name + ": $" + bal);
}
}
```

- As you can see, the Balance class is now public.
- Also, its constructor and its show( ) method are public, too.
- This means that they can be accessed by any type of code outside the MyPack package.
- For example, here TestBalance imports MyPack and is then able to make use of the Balance class:

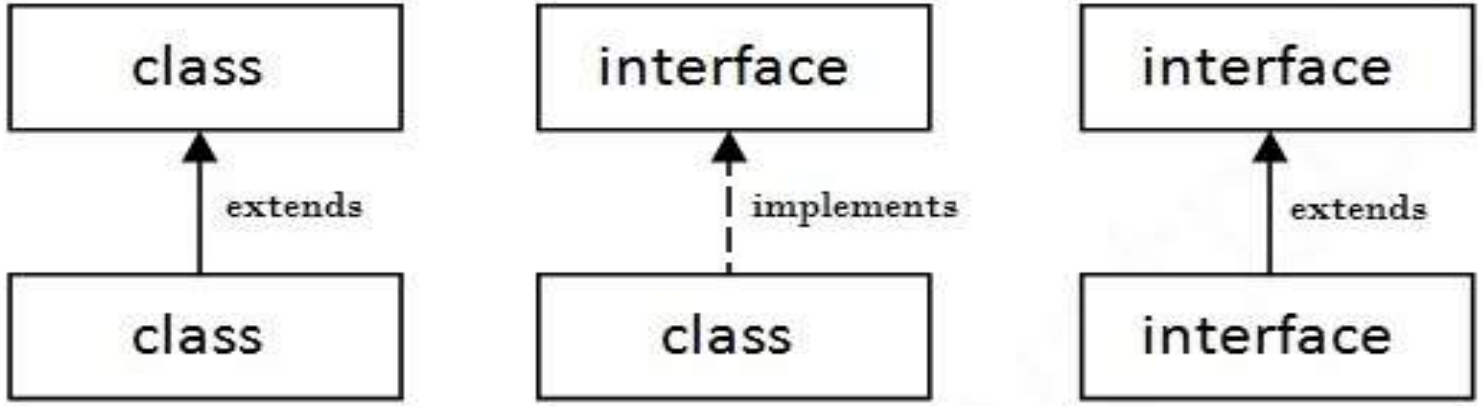
```
import MyPack.*;
class TestBalance
 {
public static void main(String args[])
 {
* Because Balance is public, you may use Balance class and call its constructor. */
Balance test = new Balance("J. J. Jaspers", 99.88);
test.show();
// you may also call show()
 }
}
```

As an experiment, remove the public specifier from the Balance class and then try compiling TestBalance. As explained, errors will result.

INTERFACES

- interfaces Using the keyword interface, you can fully abstract a class' interface from its implementation.
- That is, using interface, you can specify what a class must do, but not how it does it.
- Interfaces are syntactically similar to classes, but they lack instance variables, and, as a general rule, their methods are declared without any body.
- any number of classes can implement an interface. Also, one class can implement any number of interfaces
- Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism.

Understanding relationship between classes and interfaces



SIMPLE EXAMPLE

```
interface Drawable{
void draw();
}
//Implementation: by second user
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}
class Circle implements Drawable{
public void draw(){System.out.println("drawing circle");}
}
/Using interface: by third user
class TestInterface1{
public static void main(String args[]){
Drawable d=new Circle();//In real scenario, object is provided by method e.g. getDrawable()
d.draw();
}}
```

Multiple inheritance in Java by interface

- If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.

```
interface Printable{
void print();
}
interface Showable{
void show();
}
class A7 implements Printable,Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
A7 obj = new A7();
obj.print();
obj.show();
 }
}
```

Defining an Interface

An interface is defined much like a class.

This is a simplified general form of an interface:

```
access interface name
 {
return-type method-name1(parameter-list); return-type method-name2(parameter-list);
type final-varname1 = value;
type final-varname2 = value;
 //... return-type method-nameN(parameter-list); type final-varnameN = value;
}
```

Here is an example of an interface definition.

It declares a simple interface that contains one method called callback( ) that takes a single integer parameter.

```
interface Callback
 {
 void callback(int param);
}
```

Implementing Interfaces

- Once an **interface** has been defined, one or more classes can implement that interface.
- To implement an interface, include the **implements** clause in a class definition, and then
- create the methods required by the interface. The general form of a class that includes the **implements** clause looks like this:

class *classname* [extends *superclass*] [implements *interface* [,*interface]]*

{

// class-body

}

- methods that implement an interface must be declared **public**.

```
class Client implements Callback {
// Implement Callback's interface
public void callback(int p) {
System.out.println("callback called with " + p);
}
}
```

Notice that **callback( )** is declared using the **public** access modifier**.**

- For example, the following version of **Client** implements **callback( )** and adds the method **nonIfaceMeth( )**:

```
class Client implements Callback
 {
// Implement Callback's interface
public void callback(int p)
{
System.out.println("callback called with " + p);
}
void nonIfaceMeth()
 {
System.out.println("Classes that implement interfaces " + "may also define other members, too.");
}
}
```

Accessing Implementations Through Interface References

The following example calls the **callback( )** method via an interface reference variable:

```
class TestIface
 {
public static void main(String args[]) {
Callback c = new Client();
c.callback(42);
}
```

The output of this program is shown here:

```
callback called with 42
```

- Notice that variable **c** is declared to be of the interface type **Callback**, yet it was assigned an instance of **Client**. Although **c** can be used to access the **callback( )** method, it cannot access any other members of the **Client** class. An interface reference variable has knowledge only of the methods declared by its **interface** declaration. Thus, **c** could not be used to access **nonIfaceMeth( )** since it is defined by **Client** but not **Callback**.
- While the preceding example shows, mechanically, how an interface reference variable can access an implementation object, it does not demonstrate the polymorphic power of such a reference. To sample this usage, first create the second implementation of **Callback**,
- shown here:

```
// Another implementation of Callback.
class AnotherClient implements Callback {
// Implement Callback's interface
public void callback(int p) {
System.out.println("Another version of callback");
System.out.println("p squared is " + (p*p));
}
}
```

Now, try the following class:

```
class TestIface2 {
public static void main(String args[]) {
Callback c = new Client();
AnotherClient ob = new AnotherClient();
c.callback(42);
c = ob; // c now refers to AnotherClient object
c.callback(42);
}
}
```

The output from this program is shown here:

```
callback called with 42
Another version of callback
p squared is 1764
```

Nested Interfaces

- An interface can be declared a member of a class or another interface. Such an interface is called a *member interface* or a *nested interface*. A nested interface can be declared as **public**, **private**, or **protected**. This differs from a top-level interface, which must either be declared as **public** or use the default access level,

```
// A nested interface example.
// This class contains a member interface.
class A {
// this is a nested interface
public interface NestedIF {
boolean isNotNegative(int x);
}
}
// B implements the nested interface.
class B implements A.NestedIF {
public boolean isNotNegative(int x) {
return x < 0 ? false: true;
}
}
```

```
class NestedIFDemo {
public static void main(String args[]) {
// use a nested interface reference
A.NestedIF nif = new B();
if(nif.isNotNegative(10))
System.out.println("10 is not negative");
if(nif.isNotNegative(-12))
System.out.println("this won't be displayed");
}
}
```

- Notice that **A** defines a member interface called **NestedIF** and that it is declared **public**.Next, **B** implements the nested interface by specifying
- implements A.NestedIF
- Notice that the name is fully qualified by the enclosing class' name. Inside the **main( )** method, an **A.NestedIF** reference called **nif** is created, and it is assigned a reference to a **B** object. Because **B** implements **A.NestedIF**, this is legal.

Variables in Interfaces

- You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values.
- When you include that interface in a class (that is, when you “implement” the interface), all of those
- variable names will be in scope as constants

```
import java.util.Random;
interface SharedConstants {
int NO = 0;
int YES = 1;
int MAYBE = 2;
int LATER = 3;
int SOON = 4;
int NEVER = 5;
}
class Question implements SharedConstants {
Random rand = new Random();
int ask() {
int prob = (int) (100 * rand.nextDouble());
if (prob < 30)
```

```
else if (prob < 60)
return YES; // 30%
else if (prob < 75)
return LATER; // 15%
else if (prob < 98)
return SOON; // 13%
else
return NEVER; // 2%
}
}
class AskMe implements SharedConstants {
static void answer(int result) {
switch(result) {
case NO:
System.out.println("No");
break;
case YES:
System.out.println("Yes");
break;
case MAYBE:
System.out.println("Maybe");
break;
case LATER:
System.out.println("Later");
break;
case SOON:
System.out.println("Soon");
break;
case NEVER:
System.out.println("Never");
break;
}
}
public static void main(String args[]) {
Question q = new Question();
answer(q.ask());
answer(q.ask());
answer(q.ask());
answer(q.ask());
}
}
```

Here is the output of a sample run of this program. Note that the results are different each time it is run.

Later

Soon

No

Yes

Interfaces Can Be Extended

- One interface can inherit another by use of the keyword **extends**. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods required by the interface inheritance chain.
- Following is an example:

```
// One interface can extend another.
interface A {
void meth1();
void meth2();
}
// B now includes meth1() and meth2() -- it addsmeth3().
interface B extends A {
void meth3();
}
// This class must implement all of A and B
class MyClass implements B {
public void meth1() {
System.out.println("Implement meth1().");
}
public void meth2() {
System.out.println("Implement meth2().");
}
public void meth3() {
System.out.println("Implement meth3().");
}
}
```

```
class IFExtend {
public static void main(String arg[]) {
MyClass ob = new MyClass();

ob.meth1();
ob.meth2();
ob.meth3();
}
}
```

Static Method in Interface

```
interface Drawable{
void draw();
static int cube(int x){return x*x*x;}
}
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");
}
}
class TestInterfaceStatic{
public static void main(String args[]){
Drawable d=new Rectangle();
d.draw();
System.out.println(Drawable.cube(3));
}}
```

Difference between abstract class and interface

abstract class	Interface
1) Abstract class can have abstract and non-abstract methods.	Interface can have only abstract methods. Since Java 8, it can have default and static methods also.
2) Abstract class doesn't support multiple inheritance.	Interface supports multiple inheritance.
3) Abstract class can have final, non-final, static and non-static variables.	Interface has only static and final variables.
4) Abstract class can provide the implementation of interface.	Interface can't provide the implementation of abstract class.
5) The abstract keyword is used to declare abstract class.	The interface keyword is used to declare interface.

6)Example:
public abstract class Shape{
public abstract void draw();
}
	Example:
	
	public interface Drawable{
	void draw();
	}

```
//Creating interface that has 4 methods
interface A{
void a();//bydefault, public and abstract
void b();
void c();
void d();
}

//Creating abstract class that provides the implementation of one meth
od of A interface
abstract class B implements A{
public void c(){System.out.println("I am C");}
}

//Creating subclass of abstract class, now we need to provide the impl
ementation of rest of the methods
class M extends B{
public void a(){System.out.println("I am a");}
public void b(){System.out.println("I am b");}
public void d(){System.out.println("I am d");}
}

//Creating a test class that calls the methods of A interface
class Test5{
public static void main(String args[]){
A a=new M();
a.a();
a.b();
a.c();
a.d();
}
}
```