Introduction to Java

Material drawn from [Lewis01, Kjell00, Mancoridis01]

Java Basics

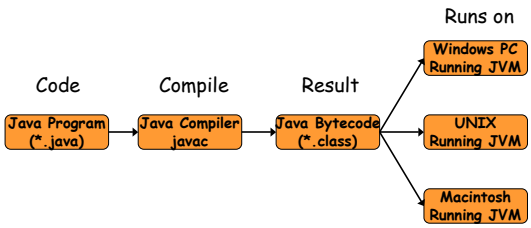
Java

- Developed by James Gosling at Sun Microsystems.
- Introduced in 1995.
- Is one the fastest growing programming technologies of all time.

Bytecode

- Java programs are translated into an intermediate language called **bytecode**.
- Bytecode is the same no matter which computer platform it is run on.
- Bytecode is translated into native code that the computer can execute on a program called the **Java Virtual Machine** (JVM).
- The Bytecode can be executed on any computer that has the JVM. Hence Java's slogan, "Write once, run anywhere".

The Java Environment



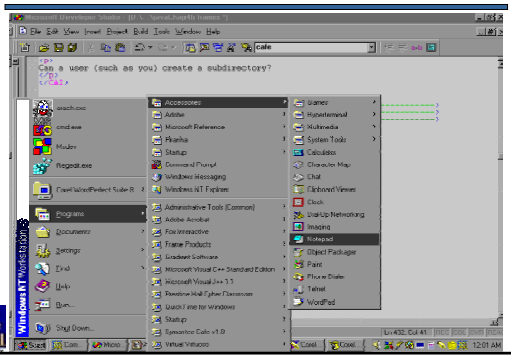
Installing Java

- The Java Development Kit (JDK) is a collection of software available at no charge from Sun Microsystems, Inc. The v1.3 download is available at java.sun.com.

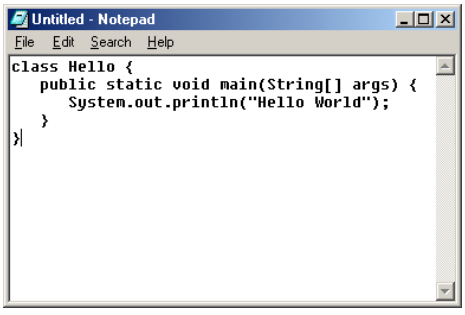
Sample Java Program

```
class Hello {

   public static void main ( String[] args ) {
      System.out.println("Hello World!");
   }
}
```

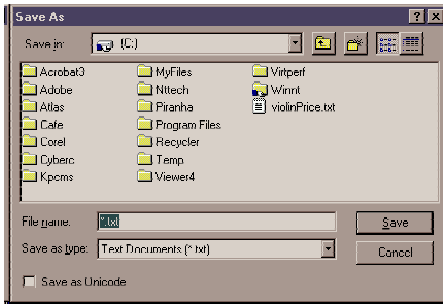
Try it yourself (Start a Notepad)



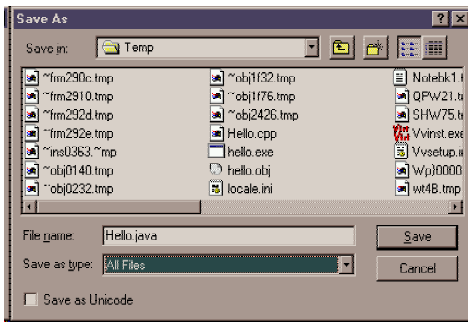
Notepad



Save the Source File

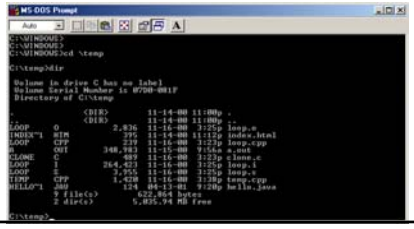


Name the File



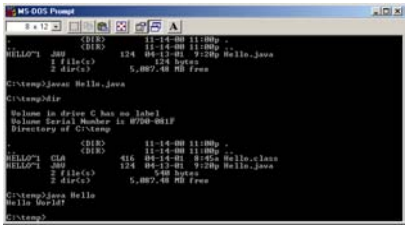
Open a Command Interpreter Window

1. Start up the DOS command prompt
2. Under the system prompt type *cd \temp* to get to the directory where you saved *Hello.java*
3. Type command *dir*, to list the files under the directory. You should see *Hello.java*



Execute Program on the Command Interpreter Window

4. Type *javac Hello.java* to compiler the source code, the Bytecode file *Hello.class* will be generated if there are no errors.
5. Type *java Hello* to run the program



Example Source Program

- The file must be named *Hello.java* to match the class name containing the main method.
- Java is case sensitive. This program defines a class called “Hello”.

Example Source Program (cont’d)

- A class is an object oriented construct. It is designed to perform a specific task. A Java class is defined by its class name, an open curly brace, a list of methods and fields, and a close curly brace.
- The name of the class is made of alphabetical characters and digits without spaces, the first character must be alphabetical. We will discuss classes in more depth later.

Between the Braces

- The line *public static void main (String[] args)* shows where the program will start running. The word *main* means that this is the *main* method – The JVM starts running any program by executing this method first.
- The *main* method in *Hello.java* consists of a single statement

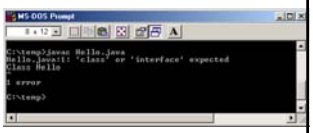
  System.out.println("Hello World!");
- The statement outputs the characters between quotes to the console.

Syntax Errors

program with a deliberate error

```
public Class Hello {

  public static void main ( String[]
    args) {
     System.out.println("Hello World!");
  }
}
```

results



- The required word "class" has been changed to "Class" with a capital "C". This is called a *syntax error*.
- A **syntax error** is a "spelling or grammatical error" in the program.
- The error message is not very clear. But at least it shows *where* the error is.
- The compiler will not create a new bytecode file because it stops compiling when it gets to an error.

Edit, Compile and Run Cycle

1. Edit the program using Notepad.
2. Save the program to the disk
3. Compile the program with the *javac* command.
4. If there are syntax errors, go back to step 1.
5. Run the program with the *java* command.
6. If it does not run correctly, go back to step 1.
7. When it runs correctly, quit.

Bugs

- Just because a program compiles and runs without complaint does not mean that it is correct.

```
public class Hello {

   public static void main ( String[] args ) {
      System.out.println("Hello Wolrd!");
   }
}
```

We expect it to generate Hello World on the screen, however, we made a mistake.

- When a program compiles without any syntax errors, but does not perform as expected when it runs, the program is said to have a **bug**.

Exercise

```
public class MyFirstProgram {

   public static void main ( String[] args )
{
      System.out.println("Hello World!");
      System.out.println("I am learning
Java");
      System.out.println("and I love it!");
   }
}
```

- Write a Java program that prints

Hello World!
I am learning Java
and I love it!

On the screen.

Layout

```
public class MyFirstProgram {

   public static void main ( String[] args ) {
      System.out.println("Hello World!");
      System.out.println("I am learning
Java");
      System.out.println("and I love it!");
   }
}
```

```
class            MyFirstProgram {
 public      static void main ( String[] args
)
  {
      System.  out.println(      "Hello
World!");
     System.out.println("I am learning
Java");
System.out.println("and I love it!"       );
  }}
```

The compiler does not "see" the two dimensional layout of the program. It regards the program as a stream of characters, one following the other.
It is much easier to read code that is laid out clearly. It is important to be neat and consistent when you create a source file.
Although the second version of the program runs correctly, it is harder for a person to understand.

Comments

```
// This program outputs “Hello World I am
// learning Java” to screen
public class MyFirstProgram {

  public static void main ( String[] args ) {
    //  the output statement starts here
    System.out.println("Hello World!");       // this statement outputs “Hello
World”
    System.out.println(“I am learning Java”);
  }
}
```

A **comment** is a note written to a human reader of a program. The program compiles and runs *exactly* the same with or without comments. Comments start with the two characters "//" (slash slash). Those characters and everything that follows them on the same line are ignored by the java compiler.

Comments (cont’d)

```
/*
   This is my first Java program
   and I am pretty excited about it.
*/
public class MyFirstProgram {

  public static void main ( String[] args ) {
    System.out.println("Hello World!");       // this statement outputs “Hello
World”
    System.out.println(“I am learning Java”);
  }
}
```

With this style of comment, everything between the two characters "/*" and the two characters "*/" are ignored by the compiler. There can be many lines of comments between the "/*" and the "*/".

Braces

```
public class MyFirstProgram {

    public static void main ( String[] args ) {
        System.out.println("Hello World!");
        System.out.println(“I am learning Java”);
    }
}
```

Another thing to notice in programs is that for every left brace { there is a right brace } that matches. Usually there will be sets of matching braces inside other sets of matching braces. The first brace in a class (which has to be a left brace) will match the last brace in that class (which has to be a *right* brace). A brace can match just one other brace. Use indenting to help the human reader see how the braces match (and thereby see how the program has been constructed).

Data types, variables and Expressions

Data Types

- A **data type** is a scheme for representing values. An example is *int* which is the integer, a data type.
- Values are not just numbers, but any kind of data that a computer can process.
- The data type defines the kind of data that is represented by a variable.
- As with the keyword *class*, Java data types are case sensitive.

Primitive Java Data Types

Data Type	Size (byte)	Range
byte	1	-128 to 127
boolean	1	true or false
char	2 (Unicode)	A-Z, a-z, 0-9, etc.
short	2	-32768 to 32767
int	4	(about)–2 million to 2 million
long	8	(about)-10E18 to 10E18
float	4	-3.4E38 to 3.4E18
double	8	-1.7E308 to 1.7E308

- There are only eight primitive data types.
- A programmer cannot create new primitive data types.

Objects

- All data in Java falls into one of two categories: **primitive data** and **objects**. There are only eight primitive data types. Any data type you create will be an object.

- An object is a structured block of data. An object may use many bytes of memory.
- The data type of an object is its **class**.
- Many classes are already defined in the Java Development Kit.
- A programmer can create new classes to meet the particular needs of a program.

More Bits for More Range

- Larger ranges of numeric values require more bits.
- Almost always you should pick a data type that has a range **much greater** than the largest number you expect to deal with.
- Note: Commas cannot be used when entering integers.

```
125   -32   58    0     45876
```

- All of the above examples are 32 bit int literals. A 64 bit long literal has a upper case 'L' or lower case 'l' at the end:

```
125l  -32l  58l   0l    45876l
```

Floating Point Types

Floating point prime data types		
Type	Size	Range
Float	32 bits	-3.4×10^{38} to3.4×10^{38}
Double	64 bits	-1.7×10^{308} to1.7×10^{308}

- In programs, floating point literals have a decimal point in them, and **no commas:**

123.0 -123.5 -198234.234 0.00000381

The char *Primitive Data Type*

- Primitive type *char* represents *a SINGLE* character.
- It does not include any font information.
- In a program, a character literal is surrounded with an apostrophe on both sides:

```
'm'   'd'   'T'
```

Primitive Data Type boolean

- It is used to represent a single true/false value.
- A boolean value can have only one of two values:

```
true   false
```

Variables

- Variables are labels that describe a particular location in memory and associate it with a data type.

Declaration of a Variable

```
public class example {

  public static void main ( String[] args ) {
    int states = 50;      // a declaration of a variable
    System.out.println("The variable states contains: " + states);
  }
}
```

- A **declaration** of a variable is where the program allocates memory for the variable.
- The declaration in the example program requested a 32 bit section of memory which will use primitive data type *int* and will be named *states*.
- A variable cannot be used in a program unless it has been declared.
- A variable can be declared only once in a particular section of code.

Syntax of Variable Declaration

- The first way to declare a variable: This specifies its data type, and reserves memory for it. It assigns zero to primitive types and null to objects.

```
dataType variableName;
```

- The second way to declare a variable: This specifies its data type, reserves memory for it, and puts an initial value into that memory. The initial value must be of the correct data type.

```
dataType variableName = initialValue;
```

Syntax of Variable Declaration (cont'd)

- The first way to declare *two* variables: all of the same data type, reserves memory for each.

```
dataType variableNameOne, variableNameTwo;
```

- The second way to declare *two* variables: both of the same data type, reserves memory, and puts an initial value in each variable.

```
dataType variableNameI = initialValueI, variableNameII=initialValueII;
```

Names of Variables

- Use only the characters 'a' through 'z', 'A' through 'Z', '0' through '9', character '_', and character '$'.
- A name cannot contain the space character.
- Do not start with a digit.
- A name can be of any reasonable length.
- Upper and lower case count as different characters. I.e., Java is case sensitive. So SUM and Sum are different names.
- A name cannot be a reserved word (keyword).
- A name must not already be in use in this block of the program.

Some Exercises

- long good-bye ; bad name: "-" not allowed
- short shift = 0; OK
- double bubble = 0,toil= 9 Missing ";" at end
- byte the bullet ; bad name: no space allowed
- int double; bad name: reserved word
- char thisMustBeTooLong ; OK, but a bit long
- int 8ball; bad name: can't start with a digit

Example Program

```
public class example {

      public static void main ( String[] args ) {
         long hoursWorked = 40;
         double payRate = 10.0, taxRate = 0.10;
         System.out.println("Hours Worked: " + hoursWorked );
         System.out.println("pay Amount : " + (hoursWorked * payRate) );
         System.out.println("tax Amount : " + (hoursWorked * payRate *
taxRate) );
      }
}
```

The character * means *multiply*. In the program, hoursWorked * payRate means to multiply the number stored in *hoursWorked* by the number stored in *payRate*. When it follows a character string, + means to add characters to the end of the character string.
Note: To *use* the value stored in a variable, just use the name of the variable.

Calculation

```
public class example {

      public static void main ( String[] args ) {
         long hoursWorked = 40;
         double payRate = 10.0, taxRate = 0.10;
         System.out.println("Hours Worked: " + hoursWorked );
         System.out.println("pay Amount : " + (hoursWorked * payRate) );
         System.out.println("tax Amount : " + (hoursWorked * payRate *
taxRate) );
      }
}
```

The parentheses around (hoursWorked * payRate) show that we want to multiply hoursWorked by payRate and then to append the result (converted to characters) to the string.
When you have a calculation as part of a System.out.println() statement, it is a good idea to surround the arithmetic part with parentheses to show that you want it done first. Sometimes this is not necessary, but it will not hurt, and makes the program more readable.

Assignment Statements

variables are expected to *vary* by having new values placed into them as the program runs. An **assignment statement** is one way to change the value of a variable.

```
public class example {

   public static void main ( String[] args ) {
      int states;       // a declaration of a variable
      states = 50;    // an assignment statement
      System.out.println("The variable states contains: " + states);
   }
}
```

The assignment statement puts the value 50 into the variable. In other words, when the program is executing there will be a 32 bit section of memory that gets the value 50.

Assignment Statement Syntax

- Assignment statements look like this:
  variableName = expression;
- The equal sign "=" means "assignment."
- *variableName* is the name of a variable that has been declared somewhere in the program.
- *expression* is an expression that has a value.
- An assignment statement asks for the computer to perform *two steps, in order:*

1. Evaluate the expression (that is: calculate a value.)
2. Store the value in the variable.

Expressions

- An **expression** is a combination of literals, operators, variables, and parentheses used to calculate a value.

  E.g. 49-x/y

- **literal:** characters that denote a value, like: 3.456
- **operator**: a symbol like plus ("+") or times ("*").
- **variable**: a section of memory containing a value.
- **parentheses**: "(" and ")".

Some Exercises

- 53 — correct
- 12 – 3) — wrong
- x + 34 — correct
- ((rate – 3) / 45 — wrong
- sum * 3 + 2 — correct
- -395.7 — correct
- (foo – 7) — correct
- Z — correct
- (x-5)/(y+6)+67 — correct
- x + — wrong

Arithmetic Operators

- An **arithmetic operator** is a symbol that performs some arithmetic.
- If several operators are used in an expression, there is a specific order in which the operations are applied.
- Operators of higher *precedence* will operate first.

Arithmetic Operators (cont'd)

Operator	Meaning	Precedence
-	Unary minus	Highest
+	Unary plus	Highest
*	Multiplication	Middle
/	Division	Middle
%	Remainder	Middle
+	Addition	Low
-	Subtraction	Low

Evaluate Equal Precedence from Left to Right

- When there are two (or more) operators of equal precedence, the expression is evaluated from left to right.

```
3 * 2 * 5
----
  6   * 5
  ------
      30
```

```
4 - 2 + 5
----
  2   + 5
  ------
      7
```

Parentheses

- Expressions within matched parentheses have the highest precedence.

```
3 *(1 + 2) * 5
   -----
3 *  3    * 5
------
   9      * 5
   ---------
          45
```

Nested Parentheses

- Sometimes, in a complicated expression, one set of parentheses is not enough. In that case use several nested sets to show what you want. The rule is:
- *The innermost set of parentheses is evaluated first.*

```
( (16/(2 * 2) ) - (4 - 8)) + 7
      -----
( (16/    4   ) - (4 - 8)) + 7
  ----------
(      4       - (4 - 8))  + 7
                   -----
(      4       -    -4  )  + 7
 ---------------------
             8             + 7
             --------------
                          15
```

Input and Output

Input and Output

- There are no standard statements in Java for doing input or output.
- All input and output is done by using *methods* found in classes within the JDK.
- Objects in memory communicate in different ways. Each way is a method, and using each method results in a different reaction to the object being used
- The *java.io* package is the package used for I/O.
- A *package* is a collection of classes which may be used by other programs.

IO Streams

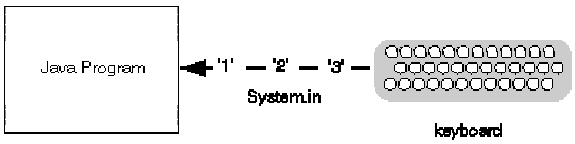
- In Java, a source of input data is called an *input stream* and the output data is called an *output stream*.

- Input data is usually called *reading* data; and output data is usually called *writing* data

Commonly used IO Streams

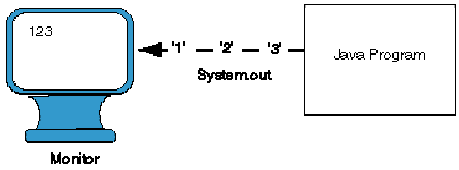
•System.in --- the input stream form the keyboard.
•System.out --- the output stream for normal results to the terminal.
•System.err --- the output stream for error messages to the terminal.

Characters In, Characters Out



The data a keyboard sends to a program is *character* data, even when the characters include the digits '0' through '9'.

Characters In, Characters Out (cont'd)

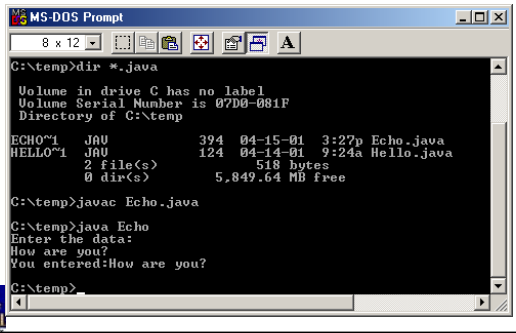


- If your program does arithmetic, the input characters will be converted into one of the primitive numeric types. Then the result is calculated (using arithmetic), and then the result is converted to character data. The information a program sends to the monitor is character data.

Example IO Program

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

        public static void main (String[] args) throws IOException {
            InputStreamReader inStream = new InputStreamReader( System.in
) ;
            BufferedReader stdin = new BufferedReader( inStream );
            String inData;
            System.out.println("Enter the data:");
            inData = stdin.readLine();
            System.out.println("You entered:" + inData );
        }
}
```

Example IO Program Results



Import a Package

- The line *import java.io.*;* says that the package *java.io* will be used. The * means that any class inside the package might be used.

Main Method

- The *main* method of class *Echo* starts with the line:

```
public static void main (String[] args) throws IOException
```

- *throws IOException* is necessary for programs that perform Input/Output. There are better ways of handling this, that will be discussed later.
- It informs the compiler that *main* performs an input operation that might fail.
- When the program is running and an input operation fails, the computer system is informed of the failure and the program halts.

Exceptions

- On 1950's mainframes and even many 1990's PC's a single input mistake could cause the entire computer system to stop ("crash").
- Java is an industrial-strength programming language, and is designed to help programmers deal with bad data and input failures.

Exceptions (cont'd)

- When an input or an output operation fails, an **exception** is generated.
- An exception is an object that contains information about the location and type of error.
- The section of code where the problem occurred can deal with the problem itself, or pass the exception on. Passing the exception on is called **throwing** an exception.

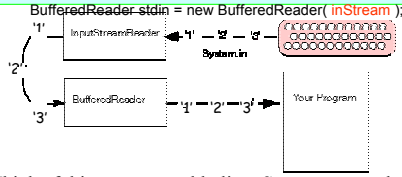
Buffered Reader

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

- This code creates a *buffered reader* that is used to read input from the keyboard.
- Note: For now, don't worry about the details of how this creates a *buffered reader*.

Assembly Line

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



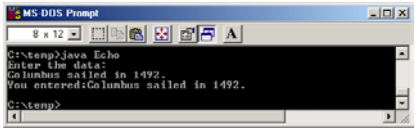
- Think of this as an assembly line: *System.in* gets characters from the keyboard. The *InputStreamReader* reads the characters from *System.in* and hands them to the *BufferedReader*. The *BufferedReader* objects hands the data to your program when requested.

readLine()

```
String inData;
System.out.println("Enter the data:");
inData = stdin.readLine();
System.out.println("You entered:" + inData );
```

- The program reads a line of character data from the buffered reader by using the method *readLine()*
- It gets a line of characters, and assign them to the String *inData*

Numeric Input



Notice that the characters '1', '4', '9', and '2' were read in and written out just as were the other characters. If you want the user to enter numeric data, your program must convert from character data to a numeric type. The characters are first read into a String object. Then the String object converts the characters into a numeric type. This is done using a *wrapper* class.

Primitive Type Wrappers

- For each primitive type, there is a corresponding *wrapper class*.
- A wrapper class can be used to convert a primitive data value into an object, and some types of objects into primitive data.

Primitive type	Wrapper type
byte	Byte
short	Short
int	Int
long	Long
float	Float
double	Double
char	Char
boolean	Boolean

Reading Integer Input

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

        public static void main (String[] args) throws IOException {
            InputStreamReader inStream = new InputStreamReader(
System.in ) ;
            BufferedReader stdin = new BufferedReader( inStream );
             String inData;
            int num, square; // declaration of two int variables
            System.out.println("Enter an integer:");
            inData = stdin.readLine();
            num = Integer.parseInt( inData ); // convert inData to int
            square = num * num ; // compute the square
            System.out.println("The square of " + inData + " is " + square );
        }
}
```

Converting to Integers

```
num = Integer.parseInt( inData ); // convert inData to int
```

- This uses the method *parseInt()* of the *Integer* wrapper class. This method takes a String containing an integer in character form.
- It looks at those characters and calculates an *int* value. That value is assigned to *num*.

Exercise

- Write a program which asks users for two integers. The two integers are added up and the result is printed to screen.

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

  public static void main (String[] args) throws IOException {
    InputStreamReader inStream = new InputStreamReader( System.in ) ;
    BufferedReader stdin = new BufferedReader( inStream );

    String line1, line2;                    // declaration of input Strings
    int    first, second, sum ;             // declaration of int variables

    System.out.println("Enter first  integer:");
    line1   = stdin.readLine();
    first   = Integer.parseInt( line1 );        // convert line1 to first int

    System.out.println("Enter second integer:");
    line2   = stdin.readLine();
    second  = Integer.parseInt( line2 );     // convert line2 to second int

    sum     = first + second;                   // add the two ints, put result in
sum

    System.out.println("The sum of " + first + " plus " + second +" is " + sum
);
  }
}
```

Converting a String to Double

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

  public static void main (String[] args) {
    final String charData = "3.14159265";
    double value;

    value  = Double.parseDouble( charData  ) ;
    System.out.println("value: " + value +" twice value: " + 2*value );
  }
}
```

- **final** means the value of *charData* can't be changed.
- The *parseDouble()* method from wrapper class *Double* examines the characters and evaluates to a *double* value.

Boolean Expressions

Relational operators

Operator	Meaning
A == B	Is A equal to B?
A < B	Is A less than B?
A > B	Is A greater than B?
A <= B	Is A less than or equal to B?
A >= B	Is A greater than or equal to B?
A != B	Is A not equal to B?

- The condition part of *if* statement is a *boolean* expression.
- A boolean expression is an expression that evaluates to *true* or *false*.
- Boolean expressions often make comparisons between numbers. A relational operator says what comparison you want to make.

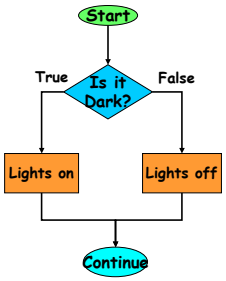
Do Lab #1 Now

Some Exercises

Expression	Value
45 == 45	true
45 <= 45	true
45 >= 4	true
4 < 8	true
43 != 43	false
45 > 45	false
45 = 45	wrong
37 <= 57	true

The if Statament

Two-way Decisions

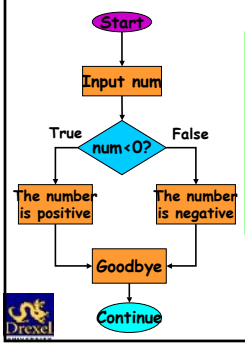


- Lights in cars are controlled with on-off switch.
- Decisions are based on question “Is it dark?”
- The answer is either true or false.

If and Else

- The words *if* and *else* are markers that divide decision into two sections.
- The *if* statement always asks a question (often about a variable.)
- If the answer is "true" only the true-branch is executed.
- If the answer is "false" only the false-branch is executed.
- No matter which branch is chosen, execution continues with the statement after the false-branch.

Flowchart of the Program




```
System.out.println("Enter an integer:");
inData = stdin.readLine();
num    = Integer.parseInt( inData );

if ( num < 0 )   // is num less than zero?
  System.out.println("The number " + num + " is
                      negative");
else
  System.out.println("The number " + num + " is
                      positive");

System.out.println("Good-by for now");
```

Complete Program

```
import java.io.*;
class NumberTester {

  public static void main (String[] args) throws IOException {
    InputStreamReader inStream = new InputStreamReader( System.in ) ;
    BufferedReader stdin = new BufferedReader( inStream );
    String inData;
    int    num;

    System.out.println("Enter an integer:");
    inData = stdin.readLine();
    num    = Integer.parseInt( inData );     // convert inData to int

    if ( num < 0 )   // is num less than zero?
      System.out.println("The number " + num + " is negative");  // true-
branch

    else
      System.out.println("The number " + num + " is positive");  // false-
branch

    System.out.println("Good-by for now");    // always executed
  }
```

More than One Statement per Branch

```
import java.io.*;
class NumberTester {

  public static void main (String[] args) throws IOException {
    InputStreamReader inStream = new InputStreamReader( System.in ) ;
    BufferedReader stdin = new BufferedReader( inStream );
    String inData;
    int    num;

    System.out.println("Enter an integer:");
    inData = stdin.readLine();
    num    = Integer.parseInt( inData );     // convert inData to int
    if ( num < 0 )   // is num less than zero? {
      System.out.println("The number " + num + " is negative");  // true-branch
      System.out.println("Negative numbers are less than 0");      // true-branch
    } else {
      System.out.println("The number " + num + " is positive");  // false-branch
      System.out.println("Positive numbers are greater than 0");  // false-branch
    }
    System.out.println("Good-by for now");    // always executed
  }
}
```

Outline of a Two-way Decision

```
…statements before the decision
if (condition) {

      ….    // true branch
} else {
    …     // false branch
}
// … statements after the decision
```

Outline of a Two-way Decision (cont'd)

- The *condition* evaluates to *true* or *false*, often by comparing the values of variables.
- The *else* divides the true branch from the false branch.
- The statement after the false branch (or false block) always will be executed.
- A block consists of several statements inside a pair of braces, { and }.
- The true branch is a block.
- The false branch is a block.
- When a block is chosen for execution, the statements in it are executed one by one.

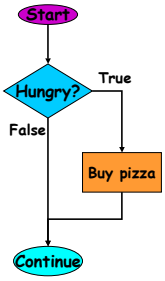
Practice

At a movie theater box office a person less than age 17 is charged the "child rate". Otherwise a person is charged "adult rate." Here is a partially complete program that does this:

Practice (cont'd)

```
import java.io.*;

class BoxOffice {

  public static void main (String[] args) throws IOException {
    InputStreamReader inStream = new InputStreamReader( System.in ) ;
    BufferedReader stdin = new BufferedReader( inStream );
    String inData;
    int    age;

    System.out.println("Enter your age:");
    inData = stdin.readLine();
    age    = Integer.parseInt( inData );     // convert inData to int

    if (      age  <  17          )
       System.out.println("Child rate.");

    else
         System.out.println("Adult rate.");

    System.out.println("Enjoy the show.");    // always executed
  }
}
```

Single-block if Statement



- There is only a *true* branch
- The if-statement always asks a question
- If the answer is *true* the true-branch is executed.
- If the answer is *false* the true-branch is skipped.
- In both cases, execution continues with the statement after the entire *if* statement.

Single-branch if

```
import java.io.*;

public class CookieDecision {

  public static void main (String[] args) throws IOException {
    String charData;
    double hunger, look;
    InputStreamReader inStream = new InputStreamReader( System.in ) ;
    BufferedReader stdin = new BufferedReader( inStream );

    System.out.println("How hungry are you (1-10):");
    charData = stdin.readLine();
    hunger   = Double.parseDouble( charData );

    System.out.println("How nice does the cookie look (1-10):");
    charData = stdin.readLine();
    look =  Double.parseDouble( charData );

    if ( (hunger + look ) > 10.0 )
      System.out.println("Buy the cookie!"  );

    System.out.println("Continue down the Mall.");
  }
}
```

Logical Operators

Logical Operators

- A *logical operator* combines *true* and *false* values into a single *true* or *false* value.
- Three types of logical operators:
  - AND (&&)
  - OR (||)
  - NOT (!)

&& Operator

Boolean expression 1 && boolean expression 2

- "&&" is the AND operator.
- If and only if both boolean expressions are true, the entire expression is true.
- If either expression (or both) is false, the entire expression is false.

Exercises

Look at the logical expression:
flour >= 4 && sugar >= 2
What will the expression give us if *flour* is 4 and *sugar* is 10?

Flour >= 4 && sugar >=2
---------- ----------
true && true
---
true

Car Rental Problem

A car rental agency wants a program to implement its rules for who can rent a car. The rules are:

- A renter must be 21 years old or older.
- A renter must have a credit card with credit of $10,000 or more.

Car Rental Problem

```
import java.io.*;

class RenterChecker {

  public static void main (String[] args) throws IOException {
    InputStreamReader inStream = new InputStreamReader( System.in ) ;
    BufferedReader stdin = new BufferedReader( inStream );
    String inData;
    int    age, credit;

    // get the age of the renter
    System.out.println("How old are you?");
    inData   = stdin.readLine();
    age    = Integer.parseInt( inData );

    // get the credit line
    System.out.println("How much credit do you have?");
    inData   = stdin.readLine();
    credit   = Integer.parseInt( inData );

    // check that both qualifications are met
    if (    age>=21 && credit>=10000    ___ )
      System.out.println("Enough to rent this car!" );
    else
      System.out.println("Have you considered a bicycle?" );
  }
}
```

|| *Operator*

Boolean expression 1 || boolean expression 2

- "||" is the OR operator.
- If either of two expressions is true, the entire expression is true.
- If and only if both expressions are false, the entire expression is false.

Exercises

Look at the logical expression:
flour >= 4 || sugar >= 2
What will the expression give us if *flour* is 2 and *sugar* is 0?

Flour >= 4 || sugar >=2
---------- ----------
false || false
---
false

Car Purchase Problem

You would like to buy a new $25,000 red Toyota Supra. To buy the car you could pay cash for it, or you could buy it on credit. Therefore, to buy this car, you need enough cash or enough credit. Write a problem to help you determine if you can buy the car.

Car Purchase Problem (cont'd)

```
import java.io.*;
class HotWheels {

  public static void main (String[] args) throws IOException {
    InputStreamReader inStream = new InputStreamReader( System.in ) ;
    BufferedReader stdin = new BufferedReader( inStream );
    String inData;
    int    cash, credit ;

    // get the cash
    System.out.println("How much cash?");
    inData   = stdin.readLine();
    cash     = Integer.parseInt( inData );

    // get the credit line
    System.out.println("How much credit do you have?");
    inData   = stdin.readLine();
    credit   = Integer.parseInt( inData );

    // check that at least one qualification is met
    if (
      cash>=25000 || credit>=25000    ) buy this car!" );
    else
      System.out.println("Have you considered a Yugo?" );
  }
}
```

More Exercises on Logical Operators

- 5 > 2 || 12 <= 7 — true
- 5 > 2 && 12 <= 7 — false
- 3 == 8 || 6!=6 — false
- 3 == 8 || !=6 — wrong

! *Operator*

- The NOT operator in Java is ! (the exclamation mark.)
- The NOT operator changes *true* to *false* and *false* to *true*, as seen in the table.

X	!X
true	false
false	true

Precedence of NOT

- It is important to put parentheses around the entire expression who's *true/false* value you wish to reverse.
- The NOT operator has high precedence, so it will be done first (before arithmetic and relational operators) unless you use parentheses.

Precedence of NOT

```
!   (Flour >= 4  || sugar >=2)
    ------------      ------------
!   (  false      ||     false  )
    ---------------------------
                ! false
                  true
```

```
!   Flour >= 4  || sugar >=2
    ----------
```

illegal! Can't use ! On an arithmetic variable

Short-circuit AND Operator

true && true	true
false && false	false
true && false	false
false && true	false

```
12 < 6  && 18 >1
-------    ----
false  &&  true
------------
   false
```

- You can get the correct answer to the question by evaluating just the first part of the expression
- Since *false* && *anything* is false, there is no need to continue after the first *false* has been encountered

How Java Works

- To evaluate *X && Y*, first evaluate X. If X is *false* then stop: the whole expression is *false*.
- Otherwise, evaluate Y then AND the two values.
- This idea is called *short-circuit evaluation*

Take Advantage of This Feature

- For example, say that two methods returning *true/false* values are combined in a boolean expression:

```
if ( reallyLongAndComplicatedMethod() &&
    simpleMethod() )
```

....

- Do we have a better arrangement?

```
if (simpleMethod() &&
    reallyLongAndComplicatedMethod())
```

....

Side-effects

- If the first clause returns *false*, the second method will not be evaluated at all, saving time.
- The result of evaluating the boolean expression will be *false* whenever *simpleMethod()* is *false*, but with this arrangement sometimes an expensive method call is skipped.
- **However** this works correctly only if the skipped method does nothing permanent.

Side-effects (cont'd)

- Short-circuit evaluation is safe when the skipped method does nothing but compute *true* or *false*. Short-circuit evaluation **is not safe** when the skipped method does more than that.
- When a method makes a permanent change to data, the method is said to have a ***side effect***.
- When methods have side effects, you must be careful when using a short-circuit operator.

Summarize of Short-Circuit Operator

true && true	Both operands are evaluated
false && true	Only first operand is evaluated
true && false	Both operands are evaluated
false && false	Only first operand is evaluated

Example

```
int count = 0;
int total = 345;

if ( count > 0 && total /count > 80)
  System.out.println("Acceptable Average");
else
  System.out.println("Poor Average");
```

- The first comparison, count > 0 acts like a guard that prevents evaluation from reaching the division when count is zero.

Short-circuit Or Operator

- The || OR operator is also a short-circuit operator.
- Since OR evaluates to *true* when one or both of its operands are *true*, short-circuit evaluation stops with the first *true*.

Precedence of Logical Operators

Operator	Precedence
!	High
&&	Medium
||	Low

- In an expression, the operator with the highest precedence will be grouped with its operand(s) first, then the next highest operator will be grouped with its operands, and so on.
- If there are several logical operators of the same precedence, they will be examined left to right.

Some Exercises

Expression	means
A || B && C	A || (B&&C)
A && B || C&& D	(A && B) || (C && D)
A && B && C || D	((A && B) && C) || D
!A && B || C	((!A) && B) || C

Increment and Decrement Operators

Increment

- In many situations, we need to add one to a variable.

```
counter = counter +1;    // add one to
    counter
```

- It is so common that we have a brief way to do it.

```
counter++;                // add one to
    counter
```

- ++ is the *increment operator*. It has the same effect as the first statement.

Increment Operator

- The increment operator ++ adds one to a variable.
- Usually the variable is an integer type, but it can be a floating point type.
- The two plus signs must not be separated by any character. (e.g., space)
- Usually they are written immediately adjacent to the variable.

Increment Operator (cont'd)

- The increment operator can be used as part of an arithmetic expression:

```
int sum = 0;
int counter = 10;

sum = counter++ ;

System.out.println("sum: "+ sum " + counter: " +
    counter );
```

Postfix Increment

- The *counter* will be incremented only **after** the value it holds has been used.
- In the example, the assignment statement will be executed in the usual two steps:
  1. Step 1: evaluate the expression on the right of the "="
     the value will be 10 (because counter has not been incremented yet.)
  2. Step 2: assign the value to the variable on the left of the "="
- sum will get 10.
- Now the ++ operator works: *counter* is incremented to 11.
- The next statement will write out: *sum: 10 counter: 11*

Use very Carefully

- The increment operator can only be used with a variable, not with a more complicated arithmetic expression. The following is **incorrect:**

```
int x = 15;
int result;
result = (x * 3 + 2)++ ;              // Wrong!
```

Prefix Increment

- The increment operator ++ can be written before a variable. When it is written before a variable (e.g., *++counter*) it is called a ***prefix*** operator; when it is written after a variable (e.g., *counter++*) it is called a ***postfix*** operator. Both uses will increment the variable; **However:**
- *++counter* means increment before using.
- *counter++* means increment after using.

Prefix Increment (cont'd)

- The increment operator can be used as part of an arithmetic expression

```
int sum = 0;
int counter = 10;

sum = ++counter ;

System.out.println("sum: "+ sum " + counter: " +
    counter );
```

Prefix Increment (cont'd)

- The *counter* will be incremented **before** the value it holds is used.
- In the example, the assignment statement will be executed in the usual two steps:
  1. Step 1: evaluate the expression on the right of the "=", the value will be 11 (because counter is incremented **before** use.)
  2. Step 2: assign the value to the variable on the left of the "=":
- sum will get 11.
- The next statement will write out: *sum: 11 counter: 11*

Sometimes it Does Not Matter

- When used with a variable alone, it doesn't matter whether you use prefix or postfix operator.
- `counter++;` is the same as `++counter;`, but `++counter;` uses less stack space.

Decrement Operator

Expression	Operation	Example	Result
x++	Add one after use	int x = 10, y; y = x++;	x is 11 y is 10
++x	Add one before use	int x=10, y; y=++x	x is 11 y is 11
x--	Subtract one after use	int x=10, y; y=x--;	x is 9 y is 10
--x	Subtract one before use	int x=10, y; y=3*x+(--x);	x is 9 y is 36

Combined Assignment Operators

- The operators +, -, *, /, (and others) can be used with = to make a combined operator

Operator	Operation	Example	Effect
=	assignment	sum = 5;	sum =5;
+=	addition with assignment	sum += 5;	sum = sum +5;
-=	subtraction with assignment	sum -= 5;	sum = sum –5;
*=	multiplication with assignment	sum *=5;	sum = sum*5;
/=	division with assignment	sum /= 5;	sum = sum/5;

Combined Assignment Operators (cont'd)

- When these operators work, the complete expression on the right of the "=" is evaluated before the operation that is part of the "=" is performed.

```
double w = 12.5 ;
double x =  3.0;

w *= x-1 ;
x -= 1 + 1;

System.out.println( " w is " + w + " x is " + x );
```

What is the output? `w is 25.0 x is 1`

Control Flow

Loops

- Computer programs can have cycles within them.
- Much of the usefulness of modern computer software comes from doing things in cycles.
- In programming, cycles are called *loops*.
- When a program has a loop in it, some statements are done over and over until a predetermined condition is met.

The while Statement

```
import java.io.*;
// Example of a while loop
public class loopExample {

  public static void main (String[] args ) {
    int count = 1;                          // start count out at one
    while ( count <= 3 ) {                  // loop while count is <= 3

      System.out.println( "count is:" + count );
      count++;                      // add one to count
    }
    System.out.println( "Done with the loop" );
  }
}
```

How the while loop works

1. The variable *count* is assigned a 1.
2. The condition (count <= 3) is evaluated as *true*.
3. Because the condition is *true*, the block statement following the while is executed.
- The current value of *count* is written out: count is 1
- *count* is incremented by one, to 2.
4. The condition (count <= 3) is evaluated as *true*.
5. Because the condition is *true*, the block statement following the while is executed.
- The current value of *count* is written out. count is 2
- *count* is incremented by one, to 3.

How the while loop works (cont'd)

6. The condition (count <= 3) is evaluated as *true*.
7. Because the condition is *true*, the block statement following the while is executed.
- The current value of *count* is written out: count is 3
- *count* is incremented by one, to 4.
8. The condition (count <= 3) is evaluated as *false*.
9. Because the condition is *false*, the block statement following the while is SKIPPED.
10. The statement after the entire while-structure is executed.
- System.out.println("Done with the loop");

Syntax of the while Statement

```
while (condition) {
    one or more statements
}
```

- The *condition* is a Boolean expression; i.e., something that evaluates to *true* or *false*.
- The *condition* can be complicated, with relational operators and logical operators.
- The *statements* are called the *loop body*.

Semantics of the while Statement

```
while (condition) {
    one or more statements
}
Statement after the loop
```

- When the *condition* is *true*, the *loop body* is executed.
- When the *condition* is *false*, the *loop body* is skipped, and the *statement after the loop* is executed.
- Once execution has passed to the *statement after the loop*, the while statement is finished.
- If the *condition* is *false* the very first time it is evaluated, the *loop body* will not be executed at all.

Three Things to Consider

There are three things to consider when your program has a loop:

1. The initial values must be set up correctly.
2. The *condition* in the while statement must be correct.
3. The change in variable(s) must be done correctly.

```
int count = 1;                          // count is initialized
while ( count <= 3 ) {                  // count is tested
  System.out.println( "count is:" + count );
  count = count + 1;                    // count is changed
}
```

Boundary Conditions are Tricky

```
int count = 1;
while ( count <= 3 ) {
  System.out.println( "count is:" +
  count );
  count = count + 1;
}
```

This loop will be executed 3 times. The output is:
count is: 1
count is: 2
count is: 3

```
int count = 1;
while ( count < 3 ) {
  System.out.println( "count is:" +
  count );
  count = count + 1;
}
```

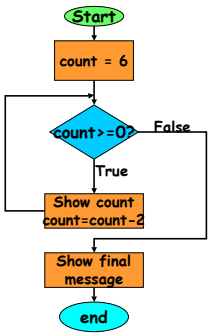
This loop will be executed 2 times. The output is:
count is: 1
count is: 2

Counting downwards by two

```
int count = 6;                          // count is initialized
while ( count >= 0 ) {                  // count is tested
  System.out.println( "count is:" + count );
  count = count - 2;                    // count is changed
 by 2
}
System.out.println( "Done counting by two's." );
```

This loop will be executed 4 times. The output is:
count is: 6
count is: 4
count is: 2
count is: 0
Done counting by two's.

Flowchart



Program to Add up User-entered Integers

•Ask user how many integers to enter.

•Write a program that will produce a sum of each integer the user enters.

Program to Add up User-entered Integers (cont'd)

```
import java.io.*;

public class addUpNumbers {

  public static void main (String[] args ) throws IOException {
    InputStreamReader inStream = new InputStreamReader(
      System.in ) ;
    BufferedReader userin = new BufferedReader( inStream );
    String inputData;
    int num;              // total numbers of integers
    int value;            // data entered by the user
    int sum = 0;          // initialize the sum

     // get the total number
     System.out.println( "Enter how many integers :" );
     inputData  = userin.readLine();
     num      = Integer.parseInt( inputData );
```

Program to Add up User-entered Integers (cont'd)

```
// get the first value
   System.out.println( "Enter first integer :" );
   inputData  = userin.readLine();
   value      = Integer.parseInt( inputData );

    while ( num != 0 ) {

       sum = sum + value;              //add value to sum
       //get the next value from the user
       System.out.println( "Enter next integer (enter 0 to
       quit):" );
        inputData = userin.readLine();
       value = Integer.parseInt( inputData );
       num = num –1;                   // refresh the counter
    }
    System.out.println( "Sum of the integers: " + sum );
  }
}
```

Sentinel Controlled Loops

- The loops we talked about so far are called *counting loops*. Loops are controlled by a counter.
- The disadvantage of the previous program is that users have to count the number of integers to be added.
- The idea of a *sentinel controlled loop* is that there is a special value (the "sentinel") that is used to say when the loop is done.

Program to Add up User-entered Integers

•Write a program that will sum each integer the user enters.
•A counting loop could do this if we knew how many integers were in the list. But users don't know this in advance.
•In this program, we use a special value (**"sentinel"**) to tell the program the loop is done. I.e., users will enter a zero to tell the program that the sum is complete.

Program to Add up User-entered Integers (cont'd)

```
import java.io.*;

public class addUpNumbers {

  public static void main (String[] args ) throws IOException {
    InputStreamReader inStream = new InputStreamReader( System.in
    ) ;        BufferedReader userin = new BufferedReader( inStream );
    String inputData;
    int value;              // data entered by the user
    int sum = 0;            // initialize the sum
```

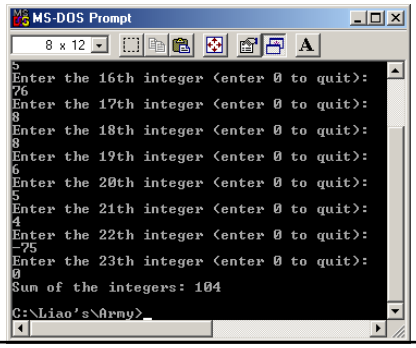
Program to Add up User-entered Integers (cont'd)

```
  // get the first value
    System.out.println( "Enter first integer (enter 0 to quit):" );
    inputData  = userin.readLine();
    value      = Integer.parseInt( inputData );

    while ( value != 0 ) {
      //add value to sum
       sum = sum + value;
      //get the next value from the user
       System.out.println( "Enter next integer (enter 0 to quit):" );
        inputData = userin.readLine();
       value = Integer.parseInt( inputData ); // "fresh" value, about to be
    }tested
    System.out.println( "Sum of the integers: " + sum );
  }
}
```

Improving the Prompt

```
int count = 0;
// get the first value
System.out.println( "Enter first integer (enter 0 to quit):" );
inputData  = userin.readLine();
value      = Integer.parseInt( inputData );

while ( value != 0 ) {
  //add value to sum
  sum = sum + value;
  //increment the count
  count = count + 1;
  //get the next value from the user
  System.out.println( "Enter the " + (count+1) + "th integer (enter 0 to
  quit):" );
  inputData  = userin.readLine();
  value      = Integer.parseInt( inputData );
}
System.out.println( "Sum of the "  + count + " integers: " + sum );
  }
}
```

Problem



Different Suffixes

- Let's change the statement that writes to the screen to this:

```
System.out.println( "Enter the " + (count+1) + suffix + " integer
    (enter 0 to quit):" );
```

- The value of *suffix* will be reference to one of the strings: "nd", "rd" or "th"
- Can a single *if-else* statement choose one of the three options?

Nested if for a Three-way Choice

- To make a three-way choice, a *nested if* is used.
- This is where an *if-else* statement is part of the true-branch (or false-branch) of another *if-else* statement.
- The nested *if* will execute only when the outer if has been evaluated.

Nested if for a Three-way Choice(cont'd)

```
String suffix;
 . . . . .   // count is the number of integers added so far.
 . . . . .   // count+1 will be the next integer to read

 if ( count+1      == 2      __ ) {

    suffix = "nd";
} else {                          // false-branch of first if
    if ( count+1      == 3      ___ ) {

        suffix = "rd";            // true-branch of 2nd if
    } else {
     suffix = "th";                  // false-branch of 2nd if
    }
}
System.out.println( "Enter the " +
                          (count+1) + suffix + " integer (enter 0 to
     quit):" );
```

How the Three-way Choice Works

- The first *if* will make a choice between its *true branch* and its *false branch*.
- Its *false branch* is complicated.
- Each branch of an *if* statement can be as complicated as needed.

Complete Add up Numbers Program

```
import java.io.*;

// Add up all the integers that the user enters.
// After the last integer to be added, the user will enter a 0.
//
public class addUpNumbers {

  public static void main (String[] args ) throws IOException {
    InputStreamReader inStream = new InputStreamReader( System.in ) ;
    BufferedReader userin = new BufferedReader( inStream );

    String inputData;
    String suffix;
    int value;              // data entered by the user
    int count = 0;          // how many integers have gone into the sum so far
    int sum   = 0;          // initialize the sum
```

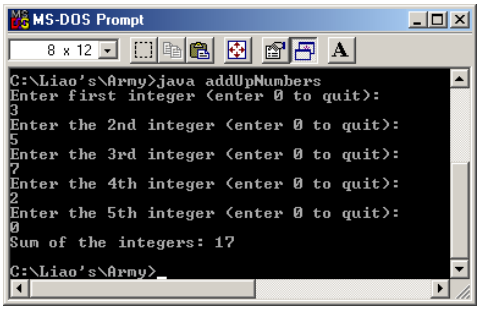
Complete Add up Numbers Program (cont'd)

```
    // get the first value
    System.out.println( "Enter first integer (enter 0 to quit):" );
    inputData  = userin.readLine();
    value      = Integer.parseInt( inputData );

    while ( value != 0 ) {
        //add value to sum
      sum   = sum + value;     // add current value to the sum
      count = count + 1;       // one more integer has gone into the sum
```

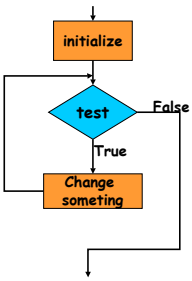
Complete Add up Numbers Program

```
// prompt for the next value
      if ( count+1  == 2  ) {
        suffix = "nd";
     } else {
        if ( count+1 == 3  ) {
          suffix = "rd";
        } else {
          suffix = "th";
        }
      }
      System.out.println( "Enter the " + (count+1) + suffix +  " integer (enter 0 to
     quit):" );
      //get the next value from the user
      inputData  = userin.readLine();
     value = Integer.parseInt( inputData );
   }
    System.out.println( "Sum of the integers: " + sum );
  }
}
```

Results



Loops (review)



1. The loop must be initialized correctly.
2. The ending condition must be tested correctly.
3. The body of the loop must change the condition that is tested.

For Statement

- Java (and several other languages) has a *for* statement which combines the three aspects of a loop into one statement. In general, it looks like this:

```
for ( initialize ; test ; change )
    {
    loopBody ;
}
```

- The *initialize*, *test*, and *change* are statements or expressions that (usually) perform the named action.
- The *loopBody* can be a single statement or a block statement.

Example

```
//   initialize      test            change
for ( count = 0; count < 10; count++ )
    {
    System.out.print( count + " " );
}
```

Is equivalent to:

```
count = 0;                          //
    initialize
while (  count < 10 ) {                 // test
  System.out.print( count + " ");
  count++ ;                         // change
}
```

Declaring the Loop Control Variable

```
int count;
for ( count = 0; count < 10; count++ )
    {
    System.out.print( count + " " );
}
```

- The declaration and initialization of *count* could be combined

```
for ( int count = 0; count < 10; count++
    ) {
    System.out.print( count + " " );
}
```

- a variable declared in a *for* statement can only be used in that statement and the body of the loop.

Declaring the Loop Control Variable (cont'd)

- The following is wrong:

```
for ( int count = 0;  count < 10; count++ ) {
    System.out.print( count + " " );
}
// NOT part of the loop body
System.out.println( "\nAfter the loop count is: " +
    count );
```

- Because the *println* statement is not part of the loop.

Do Lab#2 Now

Scope

- The ***scope*** of a variable is the block of statements where it can be used.
- The scope of a variable declared as part of a *for* statement is that statement and its loop body.
- Another way to say this is: A loop control variable declared as part of a *for* statement can only be "seen" by the *for* statement and the statements of that loop body.

Same Name Used in Several Loops

- All *for* statements may use the same identifier (the same name) in declaring their loop control variable, but each of these is a different variable which can be seen only by their own loop. Here is an example:

```
for ( int j = 0;  j < 8; j=j+2 ) {
    sumEven = sumEven + j;
}
System.out.println( "The sum of evens is: " + sumEven );

for ( int j = 1;  j < 8; j=j+2 ) {        // a different variable j
      sumOdd  = sumOdd + j;
}
```

```
public class badProgram {

  public static void main ( String[] args ) {
    int sumEven = 0;
    int sumOdd  = 0;
    int j = 0;
    int i = 1;

    for ( j = 0;  j < 8; j=j+2 )
       sumEven = sumEven + j;

    System.out.println( "The sum of evens is: " + sumEven );

    for ( i = 1;  i < 8; i=i+2 )
       sumOdd  = sumOdd + j;              // j should be i

    System.out.println( "The sum of odds is: " + sumOdd );
  }
}
```

Keep Things Local

- This program will compile and run without any error messages.
- However, the two sums are not computed correctly.
- The loops are independent. Keeping the counter variable local will make the compiler detect the error.

Conditional Operator

- To calculate an absolute value, we can do this:

```
if (value < 0)
    abs = - value;
else
    abs = value;
```

- The following statement does the same thing:

```
abs = (value < 0 ) ? -value : value
    ;
```

Tertiary Conditional Operator

```
true-or-false-condition ? value1 :
    value2
```

- The entire expression evaluates to a single value.
- That value will be one of two values.
- If the condition is *true*, then the whole expression evaluates to *value1*.
- If the condition is *false*, then the whole expression evaluates to *value2*.
- If used in an assignment statement (as above), that value is assigned to the variable.
- *value1* and *value2* must be the same type.

Practice

- Finish the following program fragment that prints the *minimum* of two variables.

```
int a = 7, b = 21;
System.out.println(
      "The min is: " + (      <      b ? a : b ) );
```

Many-way branches

- Often a program needs to make a choice among several options based on the value of a single expression.
- For example, a clothing store might offer a discount that depends on the quality of the goods:
  - Class "A" goods are not discounted at all.
  - Class "B" goods are discounted 10%.
  - Class "C" goods are discounted 20%.
  - anything else is discounted 30%.
- The program fragment is on the next slide.

switch *Statement*

```
double discount;
char   code = 'B' ;

switch ( code ) {
  case 'A':
    discount = 0.0;
    break;

  case 'B':
    discount = 0.1;
    break;

  case 'C':
    discount = 0.2;
    break;

  default:
    discount = 0.3;
}
```

- A choice is made between options based on the value in *code*.
- To execute this fragment, look down the list of cases to match the value in *code*.
- The statement between the matching *case* and the next *break* is executed. All other *case*s are skipped. If there is no match, the *default* case is chosen.

Rules of switch *statement*

```
switch ( integerExpression
    ){
  case label1 :
    statementList1
    break;

  case label2 :
    statementList2
    break;

  case label3 :
    statementList3
    break;

  . . . other cases like the
     above

  default:
     defaultStatementList
}
```

- Only one case will be selected per execution of the *switch* statement.
- The value of *integerExpression* determines which case is selected.
- *integerExpression* must evaluate to an *int* type.
- Each *label* must be an integer literal (like 0, 23), but not an expression or variable.
- There can be any number of statements in the *statementList*.

Rules of switch *statement (cont'd)*

```
switch ( integerExpression
    ){
  case label1 :
    statementList1
    break;

  case label2 :
    statementList2
    break;

  case label3 :
    statementList3
    break;

  . . . other cases like the
     above

  default:
     defaultStatementList
}
```

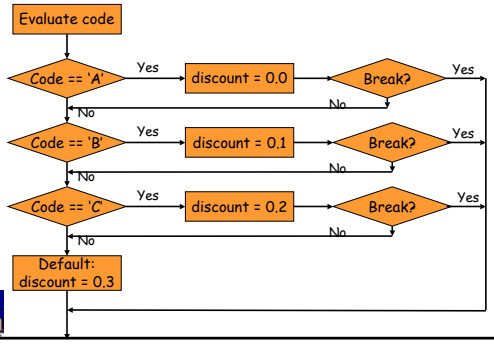
- Each time the *switch* statement is executed, the following happens:
  1. The *integerExpression* is evaluated.
  2. The *label*s after each *case* are inspected one by one, starting with the first.
  3. The body of the first label that matches has its *statementList* is executed.
  4. The statements execute until the *break* statement is encountered or the end of the *switch*.
  5. Now the entire *switch* statement is complete.
- If no case label matches the value of *integerExpression*, then the *default* case is picked, and its statements execute.
- If a *break* statement is not present, all *cases* below the selected case will also be executed.

Example

```
double discount;
char   code = 'B' ;

switch ( code ) {
  case 'A':
    discount = 0.0;
    break;

  case 'B':
    discount = 0.1;
    break;

  case 'C':
    discount = 0.2;
    break;

  default:
    discount = 0.3;
}
```

1. The *characterExpression* is evaluated.
   - In this example, the expression is just a variable, *code*, which evaluates to the character 'B'.
2. The case labels are inspected starting with the first.
3. The first one that matches is *case 'B' :*
4. The corresponding *statementList* starts executing.
   - In this example, there is just one statement.
   - The statement assigns 0.1 to *discount*.
5. The *break* statement is encountered.
6. The statement after the *switch* statement is executed.

Flow Chart of switch *Statement*



Using break

```
class Switcher {

  public static void main ( String[]
      args ) {
    char   color = 'Y' ;
    String message = "Color is";

    switch ( color ) {

      case 'R':
        message = message + " red" ;

      case 'O':
        message = message + " orange"
      ;
```

```
      case 'Y':
          message = message + "
        yellow" ;

      default:
          message = message + "
                        unknown" ;

    }
    System.out.println ( message ) ;
  }
}
```

What is the output? `Color is yellow unknown`

Using break (cont'd)

```
class Switcher {

  public static void main ( String[]
     args ){
    char   color = 'Y' ;
    String message = "Color is";

    switch ( color ) {

      case 'R':
        message = message + " red" ;
        break;

      case 'O':
            message = message + "
          orange" ;
            break;

      case 'Y':
            message = message + "
          yellow" ;
            break;

      default:
            message = message + "
                              unknown" ;

    }
    System.out.println ( message )
        ;
  }
```

What is the output? Color is yellow

Handles Upper and Lower Cases

```
import java.io.*;
class Switcher {

  public static void main ( String[] args ) throws IOException {
    String lineIn;
    char   color  ;
    String message = "Color is";

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

    System.out.println("Enter a color letter:");
    lineIn = stdin.readLine();
    color = lineIn.charAt( 0 );  // get the first character

    switch ( color ) {
```

Handles Upper and Lower Cases (Cont'd)

```
      case 'r':
      case 'R':
        message = message + " red" ;
        break;

      case 'o':
      case 'O':
        message = message + " orange" ;
        break;

      case 'y':
      case 'Y':
        message = message + " yellow" ;
        break;

      default:
          message = message + " unknown" ;
    }
    System.out.println ( message ) ;
  }
}
```

do *Statement*

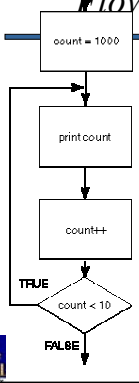
- The *do* statement is similar to the *while* statement with an important difference: the *do* statement performs a test after each execution of the loop body. Here is a counting loop that prints integers from 0 to 9:

```
int count = 0;                    // initialize count to 0

do {
  System.out.println( count ); // loop body: includes code to
  count++  ;                    // change the count
}while ( count < 10 );          // test if the loop body should be
    executed again.
```

do *Statement (cont'd)*

- All loops must do three things:
  1. The loop must be initialized correctly.
  2. The ending condition must be tested correctly.
  3. The body of the loop must change the condition that is tested.
- The *do* loop must do all three, but the ending condition test is done after the loop body has been executed.

Flow Chart of a do *Loop*



- **The body of a do loop is always executed at least once.**

Object-oriented Programming

Objected-oriented Programming

- Until now, we have not really been doing much with objects.
- The programs so far could just as easily have been written in the language C or Pascal.
- The following will discuss *objects* and *classes*, and how an object-oriented language like Java is different from a procedural language like C.

Objects

- Many programs are written to model problems of the real world.
- It is convenient to have "software objects" that are similar to "real world objects." This makes the program and its computation easier to think about.
- Object-oriented programming solve a problem by creating objects that solve the problem.

Class

- A group of objects with similar properties and behaviors is described by a **class**.
- Class is a pattern for an object.
- A class contains *fields* and *methods* that define the behavior of the object.
- A **class** is a description of a kind of object.
- A programmer may define a class using Java, or may use predefined classes that come in class libraries such as the JDK.

Objects

- In object-oriented programming, programmers use a programming language (such as Java) to describe the program as various objects. When the program is run, the object are created, and they start "doing things" by running their methods.

Objects (cont'd)

- The methods do not run in any order. For an application, the first method to run is the method named *main*. There should be only one method named *main* in an application.
- In a small application, *main* might do by itself all the computation that needs to be done. In a larger application, *main* will create objects and use their methods.

Class and Object

- A class is merely a template for one or more objects. It describes an object's behavior and state.
- It does not create any objects.
- When a programmer wants to create an object the *new* operator is used with the name of the class.
- Creating an object is called *instantiation*.

Program that instantiates an object

```
class stringTester {

  public static void main ( String[] args ) {
    String str1;                          // str1 is a variable that refers to an
     object,
                                          // but the object does not exist yet.
    int    len;                           // len is a primitive variable of type
     int

    str1 = new String("Random Jottings");  // create an object of type String

    len  = str1.length();                 // invoke the object's method
     length()

    System.out.println("The string " + strl + " is "+ len + " characters
     long");
  }
}
```

Program that instantiates an object (cont'd)

- When it is executed (as the program is running), the line

  ```
  str1 = new String("Random Jottings");
  ```

  creates an object by following the description in class *String*. The class *String* is defined in the package *java.lang* that comes with the Java system.
- The computer system finds a chunk of memory for the object, lays out this memory according to the plan (the definition of String), and puts data and methods into it.

Program that instantiates an object (cont'd)

- The data in this String object will be the characters "Random Jottings" and the methods will be all of the methods that the String class has. One of these methods is *length()*.
- The variable *str1* is used to refer to a string object. In other words, *str1* gives the object a name, like a variable.

Objects

- A variable that can refer to an object does not always have an object to refer to.
- In our program the variable *str1* refers to an object only after the *new* operator creates one.
- Before the *new* operator did its work, *str1* was a "place holder" that did not actually refer to any object.
- After the *new* operator creates the object, *str1* is used to refer to the object.

Using a Reference to an Object

- Once the object has been created (with the *new* operator), the variable *str1* refers to that object.
- That object has several methods, one of them is the *length()* method.
- A String object's *length()* method counts the characters in the string, and returns that amount.

Invoking an Object's Method

- An object consists of both variables and methods. Both of these are called "members" of the object. Java uses "dot notation" for both:

```
referenceToAnObject.memberOfObject
```

- To invoke the *length()* method of the object named *str1* the following is used:

```
len = str1.length();
```

- Remember that all method names will have "()" at their end. The above method evaluates to an integer, 15, which is assigned to the *int* variable *len*.

Constructor

- An object is a section of memory (with variables and methods), and a class is a description of objects.
- The *new* operator calls a class' constructor method.

Constructor (cont'd)

- A *constructor* has the same name as the class.

```
str1 = new String("Random Jottings");
```

- *new* creates a new object of type String. It is followed by the name of a constructor. The constructor *String()* is part of the definition for the class String.
- Constructors often are used with values (called parameters) that are to be stored in the data part of the object that is created. In the above program, the characters *Random Jottings* are stored in the data section of the new object.
- Constructors do not return anything.

Primitive Data Types and Classes

- Other than the primitive data types, *all of the other data in a Java program are represented as objects*.
- Every object in Java is an instance of a class. (In other words: the data type of an object is a *class*.)

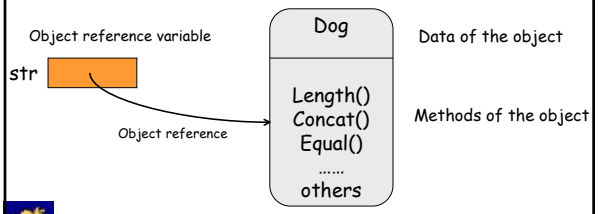
A Primitive Variable

```
long value;
value = 46823;
```

- With primitive data types, a variable is a section of memory reserved for a value of a particular style.
- 64 bits of memory are reserved for variable *value*, and 46823 is put in the memory.
- *value* references to this memory.

A String *Object*

```
String str;
str = new String("Dog");
```



Object Reference

```
String str;    // name for a future object

str =    new String( "Dog" );
```

1. An object is created using the constructor.
   The Java system keeps track of how to find the object.
2. The way to find the object is stored in the variable str.

- An ***object reference*** is information on how to find a particular object. The object is a chunk of main memory; a reference to the object is a way to get to that chunk of memory.

Object Reference (cont'd)

- The variable *str* does not actually contain the object, but contains information about where the object is located.
- Objects are created while a program is running. Each object has a unique object reference, which is used to find it.
- When an object reference is assigned to a variable, that variable says how to find that object.

Variables

- There are only primitive variables and object reference variables, and each contains a specific kind of information:

Kind	Information it contains	When on the left of "="
Primitive variable	Actual data	Previous data is replaced by new one
Reference variable	Information how to find an object	Old reference is replaced by new one

Example

```
value = 32912;
str   = new String( "Dog" );
```

- In the first statement, *value* is a primitive variable, so the assignment statement puts the data directly into it.
- In the second statement, *str* is an object reference variable, so a *reference* to the string object is put into that variable.

Two Types of Use

```
value = 32912;
str   = new String( "Dog" );
System.out.println(str);
System.out.println(value);
```

- When the statement System.out.println(str); is executed, the object referred to by *str* is found and its data is written to the monitor.
- When the statement System.out.println(value); is executed, the primitive value in the variable *value* is used directly.

Summary

1. Each time the ***new*** operator is used, a new object is created, and memory is allocated to it.
2. Each time an object is created, there is a **reference** to it.
3. This reference is saved in a variable.
4. Later on, the reference to the variable is used to find the object.
5. If another reference is saved in the variable, it replaces the previous reference.
6. If no variables hold a reference to an object, there is no way to find it, and it becomes "garbage."

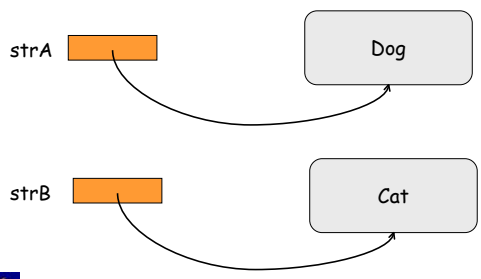
Example

```
public class egString3 {

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

    str   = new String("Dog");        // create the 1st object referenced by str
    System.out.println(str);          // follow the reference in str to the 1st
     object,
                                      // get data and print it.

    str   = new String("Cat");        // create the 2nd object referenced by str
    System.out.println(str);          // follow the reference in str to the 2nd
     object,
                                      // get data and print it.
  }
}
```

Several Objects of the Same Class

```
class egString4 {

  public static void main ( String[] args ) {
    String strA;  // reference to the first object
    String strB;  // reference to the second object

    strA   = new String( "Dog" );     // create the first object and
                                      // Save its reference.
    System.out.println(  strA  );     // follow reference to first
                                      // object and println its data.
    strB   = new String( "Cat" );     // create the second object and
                                      // Save its reference.
    System.out.println(  strB  );     // follow reference to
     second
                                      //  object and println its
     data.
    System.out.println(  strA  );     // follow reference to first
                                      //  object and println its
     data.
     }
```

Several Objects of the Same Class (cont'd)



Equality of Reference Variables

```
String strA;                          // reference to the first
 object
String strB;                    // reference to the second object

strA   = new String( "Dog" );    // create the first object and
                                 // save its reference
System.out.println( strA   );

strB   = new String( "Cat" );    // create the second object and
                                 // save its reference
System.out.println( strB   );

if ( strA == strB )
  System.out.println( "This will not print.");
```

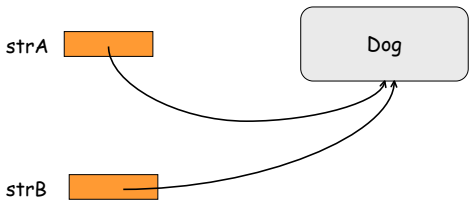
The == operator is used to look at *the contents* of two reference variables rather than the data they refer to!

Two Reference Variables Pointing to one object

```
String strA;                          // reference to the object
String strB;                          // another reference to the
 object

strA   = new String( "Dog" );    // Create the only object.
                                 // Save its reference in
 strA.
System.out.println( strA   );

strB   = strA;                        // Copy the reference to
 strB.
```

Only one object is created by new. The statement strB = strA copies the value (reference) in *strA* to *strB*. It does not make a copy of the object!

Two Reference Variables Pointing to one object (cont'd)



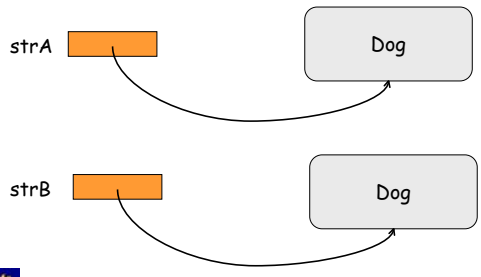
Now strA == strB will evaluate to *true*, because they contain the same reference.

Two Objects with Same Contents

```
String strA;                                  // reference to the first
 object
String strB;                                  // reference to the
 second object

strA   = new String( “Dog" );              // create the first object.
                                                  // Save its reference
System.out.println( strA   );

strB   = new String( "Dog" );              // create the second object.
                                                  // Save its reference
System.out.println( strB   );

if ( strA == strB )
```

Two objects are completely separate. They just happen to have same contents, but they consists of different chunk of memory with different references. strA == strB evaluates to *false*.

Two Objects with Same Contents (cont'd)



The equals() method

```
String strA;  // first object
String strB;  // second object

strA   = new String( "The Dog" );
strB   = new String( "The Dog" );

if ( strA.equals( strB ) )
  System.out.println( "This WILL print.");
```

The *equals(String)* method DOES look at the contents of objects. It returns either *true* or *false*.

The Class Point

- Java comes with a library of pre-defined classes that are used for building graphical user interfaces.
- This library is called the *Application Windowing Toolkit*, or the *AWT*.
- One of the many classes defined in this library is the class *Point*.
- A class library is a collection of classes that your program might use when it creates objects.

Class Description

- The class *Point* will describe two things: the variables (the data) and the methods (the behavior) that Point objects will have.

Class Description (cont'd)

```
public  class  java.awt.Point   {

   // Fields
   public int x;
   public int y;

   // Constructors
   public Point();                    // creates a point at (0,0)
   public Point(int  x, int  y);      // creates a point at (x,y)
   public Point( Point pt );          // creates a point at the location given
    in pt

   // Methods
   public boolean equals(Object  obj);     // checks two point objects for
    equal data
   public void move(int  x, int  y);       // changes the (x,y) data of a
    point object
   public String toString();               // returns character data that
    can be
                                           // printed
```

Multiple Constructors

- Any of the three constructors can be used to instantiate a Point. It is a matter of convenience which constructor you use.

```
import java.awt.*;          // import the class library where Point is
    defined
class pointEg1 {

  public static void main ( String arg[] ) {
    Point a, b, c;              // reference variables

    a = new Point();            // create a Point at ( 0,  0); save the reference in
     "a"
    b = new Point( 12, 45 ); // create a Point at (12, 45); save the reference
     in "b"
    c = new Point( b );         // create a Point at (12, 45); save the reference
     in "c"
  }
}
```

Multiple Constructors (cont'd)

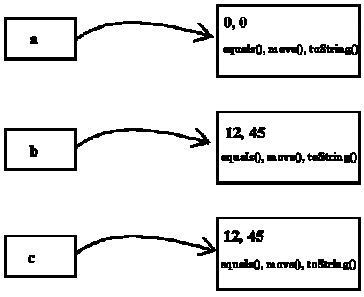
1. Declares three reference variables *a*, *b*, and *c*, which may refer to objects of type Point.
2. Instantiates a Point object with x=0 and y=0.
   (The documentation tells us that a constructor without parameters initializes x and y to zero.)
3. Saves the reference in the variable *a*.

Multiple Constructors

4. Instantiates a Point object with x=12 and y=45.
5. Saves the reference in the variable *b*.
6. Instantiates a third Point object.
7. Saves the reference in the variable *c*.

- Once each Point object is instantiated, it is the same as any other (except for the values in the data). It does not matter which constructor was used to instantiate it.

Picture of the Objects



The toString() Method

- The class *Point* provides a method called *toString()*.

```
public String toString();   // returns character data that can be printed
```

this is the method name. It takes no parameters.

this says that the method returns a String object

anywhere you have a Point object, you can use this method

Program that Uses the Method

```
import java.awt.*;

class pointEg2 {

  public static void main ( String arg[] ) {
    Point a, b, c;              // reference variables

    a = new Point();            // create a Point at ( 0,  0); save the reference in
     "a"
    b = new Point( 12, 45 ); // create a Point at (12, 45); save the reference in
     "b"
    c = new Point( b );         // create a Point at (12, 45); save the reference in
     "c"

    String strA = a.toString();     // create a String object based on the data
                                    // found in the object referenced by "a".
    System.out.println( strA );
  }
}
```

Program that Uses the Method (cont'd)

- When this program runs, the statement: *String strA = a.toString();* creates a String object based on the data in the object referred to by *a*.
- The reference variable *strA* keeps track of this new String object. Then the characters from the String are sent to the monitor with *println*. The program prints out: java.awt.Point[x=0,y=0]
- The Point object has not been altered: it still exists and is referred to by *a*. Its method *toString()* created the characters that were printed out and stored them in a String object.

Using a Temporary Object

```
import java.awt.*;

class pointEg3 {

  public static void main ( String arg[] ) {
    Point a = new Point();            // declarations and construction
     combined
    Point b = new Point( 12, 45 );
    Point c = new Point( b );

    System.out.println( a.toString() ); // create a temporary String based
     on "a"
  }
}
```

Using a Temporary Object (cont'd)

1. When the statement executes, *a* refers to an object with data (0,0).
2. The toString() method of that object is called.
3. The toString() method creates a String object and returns a reference to it.
4. The println method of System.out uses the reference to find the data to print out on the monitor.
5. The statement finishes execution; the reference to the String has not been saved anywhere.

Automatic Call of toString()

- System.out.println() can deal with several types of parameters, and one of them is *reference to a String*.
- When a parameter should be a String reference, but a reference to another type of object is given, Java will call the object's *toString()* method to get a String reference.

```
Point a = new Point();  // a is a Point reference

System.out.println( a );
```

Changing Data Inside a Point

- One of the methods is:
  public void move(int x, int y) ;
- This method is used to change the x and the y data inside a Point object. The modifier *public* means that it can be used anywhere in your program; *void* means that it does not return a value.
- This part of the description (int x, int y) says that when you use *move*, you need to supply two *int* parameters that give the new location of the point. A *parameter* is information you supply to a method.

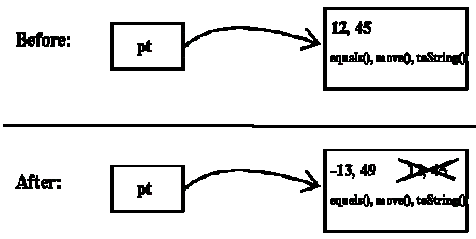
Changing Data Inside a Point (cont'd)

```
import java.awt.*;
class pointEg4 {

  public static void main ( String arg[] ) {
    Point pt = new Point( 12, 45 );              // construct a Point
    System.out.println( "First values: " + pt );

    pt.move( -13, 49 ) ;                         // change the x and y in the
     Point
    System.out.println( "Final values: " + pt );

  }
}
```

Here is what it writes to the screen:

```
First values: java.awt.Point[x=12,y=45]
Final values: java.awt.Point[x=-13,y=49]
```

One Object, with Changing Data



The same reference variable, pt, points to the same object, but its data has been changed.

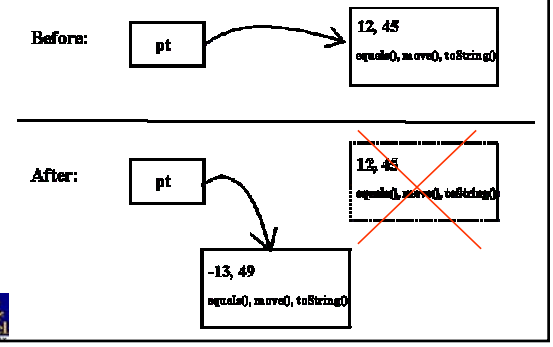
Similar Program

```
import java.awt.*;
class pointEg4 {

  public static void main ( String arg[] ) {
    Point pt = new Point( 12, 45 );              // construct a Point
    System.out.println( "First values: " + pt );

    pt       = new Point( -13, 49 ) ;            // construct a new Point
    System.out.println( "Final values: " + pt );

  }
}
```

Here is what it writes to the screen:

First values: java.awt.Point[x=12,y=45]
Final values: java.awt.Point[x=-13,y=49]

The First Object Becomes Garbage



null

- A reference variable can hold information about the location of an object, but does not hold the object itself. This following code declares two variables but does not construct any objects.

```
String a;
Point  b;
```

- The following constructs objects and puts references in the variables:

```
a = "Elaine the fair." ;
b = new Point( 23, 491
              );
```

null *(cont'd)*

- There is a special value called **null** that can be assigned to any object reference variable.
- The value *null* is a special value that means "no object." A reference variable is set to *null* when it is not referring to any object.

Testing for null

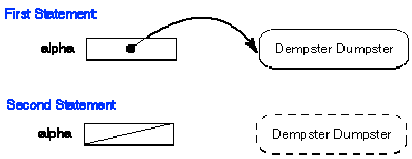
```
public class nullDemo1 {

 public static void main (String[] arg) {
  String a = "Random Jottings";  // 1.  an object is created;
                                 //     variable "a" refers to it
  String b = null;                     // 2.  variable "b" refers
   to no object.
  String c = "";                       // 3.  an object is created
   (containing
                                       //      no characters);
   variable c refers
                                 //     to it.
  if ( a != null )                     // 4.  ( a != null ) is true,
   so
    System.out.println( a );           // 5.  the println( a )
   executes.

  if ( b != null )                     // 6.  ( b != null ) is false,
   so
    System.out.println( b );           // 7.  the println( b ) is
   skipped.
```

Garbage

```
String alpha = "Dempster Dumpster";
alpha = null;
```

- The first statement does two things: (1) a String object is created, containing the characters "Dempster Dumpster". Then, (2) a reference to that object is saved in the reference variable *alpha.*
- The second statement assigns the value *null* to *alpha*. When this happens, the reference to the object is lost. Since there is no reference to the object elsewhere, it is now garbage.

Garbage (cont'd)



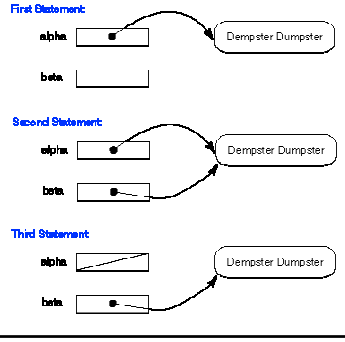
- The object still exists in memory. The memory it consists of will eventually be recycled by the *garbage collector*, and will be made available for new objects.

Not Garbage

```
String alpha = "Dempster Dumpster";
String beta  = alpha;
alpha = null;
```

- This time, the object does not become garbage. This is because in the second statement (a reference to the object) is saved in a second variable, *beta.* Now when, in the third statement, *alpha* is set to *null*, there is still a reference to the object.
- There can be many copies of the reference to an object. Only when there is no reference to an object anywhere does the object become garbage.

Not Garbage (cont'd)



The concat() *Method*

```
String name = first.concat(  last );
```

- a String reference parameter
- the name of the method
- dot notation used to call an object's method.
- the method returns a reference to a new String object

- The *concat* method performs string *concatenation*. This is constructing a new String by using the data in two other Strings.

The concat() *Method (cont'd)*

```
String first = "Kublai" ;
String last  = " Khan" ;
String name  = first.concat( last );
```

- In the example, the first two strings (referenced by *first* and *last*) supply the data that *concat* uses to construct a third string (referenced by *name*.)
- The first two Strings are NOT changed by the action of *concat*. A new string is constructed that contains the results of the action we wanted.

+ Operator

```
String first = "Kublai" ;
String last  = " Khan" ;
String name  = first + last ;
```

The "+" operator is a short way of asking for concatenation. (If result is a number, it is converted into characters before the concatenation is done).

This does the same thing as the first version. Only Strings have short-cut ways of doing things.

The trim() Method

```
String userData = "     745   ";
String fixed;

fixed = userData.trim();
```

The trim() method creates a new String. The new String contains the same characters as the old one but has any *whitespace* characters (blanks, tabs, spaces, etc.) removed from both ends (but not from the middle).
The new String referenced by *fixed* will contain the characters "745" without the surrounding spaces. This is often useful in preparing user input data for conversion from character to numeric form.

The startsWith() Method

```
public boolean startsWith(String prefix);
```

The *startsWith()* method tests if one String is the prefix of another. This is frequently needed in computer programs.

```
public class prefixTest {
  public static void main ( String args[] ) {
    String burns = "No pain, no gain.";

    if ( burns.startsWith( "No pain" ) )
      System.out.println( "Prefix 1 matches." );
    else
      System.out.println( "Prefix 1 fails." );

    if ( burns.startsWith( "no pain" ) )
      System.out.println( "Prefix 2 matches." );
    else
      System.out.println( "Prefix 2 fails." );

    if ( burns.startsWith( "  No pain" ) )
      System.out.println( "Prefix 3 matches." );
    else
      System.out.println( "Prefix 3 fails." );

    if ( burns.startsWith( "  No pain".trim() ) )
      System.out.println( "Prefix 4 matches." );
    else
      System.out.println( "Prefix 4 fails." );
  }
}
```

Output

```
Prefix 1 matches.
Prefix 2 fails.
Prefix 3 fails.
Prefix 4 matches.
```

Object-oriented Programming

- Object-Oriented programming (in Java or in any object-oriented language) consists of two big steps:

1. Creating the Program.

- The programmer defines classes that describe future objects that the program will use when it is running.
- The programmer defines a class that contains the static *main()* method which is used to start the program running.

Object-oriented Programming in Java

2. Running the Program.

- The program is compiled into bytecode.
- The java interpreter looks for a static *main()* method and starts running it.
- As the program runs, objects are created and their methods are activated.
- The program does its work by creating objects and activating their methods.
- The exact order of object creation and method activation depends on to task to be performed and the input data.

Define a Class

- Class definitions look like this:

```
class ClassName {

  Descriptions of the variables and methods and the
  constructors that initialize a new object.

}
```

Define a Class

- Often programmers separate the definition into three sections:

```
class ClassName {
  // Description of the variables.

  // Description of the constructors.

  // Description of the methods.
}
```

- Separation into sections is done for convenience; it is not a rule of the language.

An Example

```
class HelloObject {

  // method definition
  void speak() {
     System.out.println("Hello from an object!");
  }
}

class HelloTester {

  public static void main ( String[] args ) {

    HelloObject anObject = new HelloObject();

    anObject.speak();
  }
}
```

An Example (cont'd)

- The definition of class HelloObject only defines a method for the class.
- When the main() method starts, it constructs a HelloObject and then invokes its speak() method.
- Objects of this class have no instance variables. The class does have a constructor.

An Example (cont'd)

- Save the file as HelloTester.java because the file name must match the name of the class that contains the main() method. When you run this program it will write this out:

Hello from an object!

Method Definition

Method definitions look like this:

```
returnType methodName( parameterList ) {
  // Java statements

  return returnValue;
}
```

- The *returnType* is the type of value that the method hands back to the caller of the method.
- Your methods can return values just as do methods from library classes. The **return** statement is used by a method to hand back a value to the caller.

void

- If you want a method that does something, but does not hand a value back to the caller, use a return type of **void** and do not use a **return** statement.
- The method will automatically return to the caller after it executes. Here is the method from the example program:

```
// method definition
  void speak() {
    System.out.println("Hello from an object!");
  }
```

```
class HelloObject  {                        // 2a.  The class definition is
                                            //      used to make the object

  void speak()  {                           // 2b.  A speak() method
                                            //      is included in the object.

    System.out.println("Hello from an object!"); // 3a.  The speak() method
                                                 //   of the object prints on
    the screen.
                                                 // 3b. The method returns
    to the caller.
  }
}

class HelloTester {
  public static void main ( String[] args ) {     // 1.  Main starts running.

    HelloObject anObject = new HelloObject();     // 2.  A HelloObject
                                                  //     is created.

    anObject.speak();                             // 3.  The object's speak()
                                                  //     method is called.

  }                                               // 4.  The entire program is
finished.
```

Picture of the Program

- The *speak()* method is part of an object. It only exists when there is an object for it.
- The variable *anObject* is a reference to the object that was created. That object contains the method *speak()*. So *main()* can find and activate *speak()* with:

```
anObject.speak();
```

Constructor

- You will need to include a constructor in the class definition when you need to pass some data to the object being instantiated.

Default Constructor

- There is no constructor described in the class definition of HelloObject. This is OK.
- All classes have a constructor even if one is not included explicitly in the class definition.
- A default constructor will be supplied automatically.

Problem

- So far, the *speak* method of the *HelloObject* class always prints the same string to the monitor.
- If different objects of the *HelloObject* class are to print different things, then each instance will have its own unique data, which will have to be initialized. This calls for a constructor.

Improved Program

```
class HelloObject {
  String greeting;      // keep the object's message here


  void speak() {
    System.out.println( greeting );    // print the object's message
  }
}
```

- The class is not complete because there is not yet a way to initialize the greeting

Constructor Definition Syntax

```
className( parameterList ) {
  Statements usually using the variables of the
  class and the parameters in the parameterList.
}
```

- No return type is listed before the *className*.
- There is no return statement in the body of the constructor.
- The constructor must have the same name as the class. The *parameterList* is a list of values and their types that will be handed to the constructor when it is asked to construct a new object.

Constructor Definition Syntax (cont'd)

- Parameter lists look like this (the same as parameter lists for methods):

TypeName1 parameterName1,
TypeName2 parameterName2, ...

- It is OK to have an empty parameter list.
- A class often has several constructors with different parameters defined for each constructor.

Constructor Definition Syntax (cont'd)

- A constructor returns a reference to the object it constructs. You do not use a *return* statement for this to happen. Usually the programmer will save the reference in a variable.

Coding a Constructor

```
public class HelloObject {
  String greeting;

  HelloObject( String st ) {
      greeting = st;
  }

  void speak() {
    System.out.println( greeting );
  }
}
```

The constructor will initialize the variable *greeting* with data that is supplied when the constructor is used.

```
public class HelloObject  {                 // 3a.  the class definition is
                                            //      used to construct the
    object.
 String greeting;

 HelloObject( String st ) {                 // 3b.  the constructor is used to
   greeting = st;                           //      initialize the variable.
 }

 void speak() {
   System.out.println( greeting );
 }
}

class HelloTester {

 public static void main ( String[] args ){ // 1.  main starts running

   HelloObject anObject =                   // 2.  the String "A
    Greeting"
     new HelloObject("A Greeting!");        //     is constructed.
                                            // 3.  a HelloObject is
    created. A
                                            // reference to the String is
    passed
                                            // to the constructor
```

The private *Visibility Modifier*

- When a member of a class is declared **private.** It can be used only by the methods of that class. Not all objects can "see" all private variables.
- Here is a checking account class definition with each of its variables declared to be **private**.
- Only the methods of a CheckingAccount object can "see" the values in *accountNumber, accountHolder,* and *balance*

```
public class CheckingAccount {
  // data-declarations
  private String accountNumber;
  private String accountHolder;
  private int    balance;

  //constructors
  CheckingAccount( String accNumber, String holder, int start ) {
    accountNumber = accNumber ;
    accountHolder = holder ;
    balance       = start ;
  }

 // methods
  int currentBalance() {
    return balance ;
  }

  void  processDeposit( int amount ) {
    balance = balance + amount;
  }
}
```

Accesser Methods

- A class with private data controls access to that data by using an **accesser methods**.
- An accesser method is a method which uses the private data of its class and is visible to other classes. Some accessr methods alter data; others return a value but don't alter data.

Accesser Methods (cont'd)

```
public class CheckingAccount {
  private String accountNumber;
  private String accountHolder;
  private int    balance;
// . . . .
}

class CheckingAccountTester {
  public static void main( String[] args ) {
    CheckingAccount bobsAccount = new CheckingAccount( "999", "Bob",
     100 );

    System.out.println( bobsAccount.currentBalance() );
    bobsAccount.processDeposit( 200 );
    System.out.println( bobsAccount.currentBalance() );
  }
}
```

main() *Can't See Private Data*

```
class CheckingAccount
{
  private String accountNumber;
  private String accountHolder;
  private int    balance;
// . . . .
}

class CheckingAccountTester
{
  public static void main( String[] args )
  {
    CheckingAccount bobsAccount = new CheckingAccount( "999", "Bob",
     100 );

    System.out.println( bobsAccount.balance );
    bobsAccount.balance = bobsAccount.balance + 200;  // cannot do this
    System.out.println( bobsAccount.balance );
  }
}
```

```
compiling: CheckingAccountTester.java
CheckingAccountTester.java:46:
Variable balance in class CheckingAccount not accessible from class
CheckingAccountTester.
        System.out.println( bobsAccount.balance );
                            ^
CheckingAccountTester.java:47:
Variable balance in class CheckingAccount not accessible from class
CheckingAccountTester.
        bobsAccount.balance = bobsAccount.balance + 200;
        ^
CheckingAccountTester.java:47:
Variable balance in class CheckingAccount not accessible from class
CheckingAccountTester.
        bobsAccount.balance = bobsAccount.balance + 200;
                              ^
CheckingAccountTester.java:48:
Variable balance in class CheckingAccount not accessible from class
CheckingAccountTester.
        System.out.println( bobsAccount.balance );
                            ^
4 errors
```

Careful Access Control

- It may seem a bit silly that the *CheckingAccount* class used *private* to prevent *main()* from seeing its variables, and then it provided some methods so that *main()* could get at them anyway. But the idea is that the access methods could check each access to the private data.
- For example, a programmer can't increase the balance of a checking account by writing:

```
bobsAccount.balance = bobsAccount.balance + 200;
```

Careful Access Control (cont'd)

- To increase the balance, the *processDeposit()* method must be used, which in a more elaborate program might check that the account is not frozen, might insist on a password before it changes anything, and might write a log file of every change.

Careful Access Control (cont'd)

- By declaring data to be private and by forcing the use of access methods, it is easier to keep objects consistent and bug-free.
- This is somewhat like putting all the electronics of a TV set inside of a box, and allowing the user to change things only by using the controls on the outside of the box. TV sets would not last long if users customarily changed channels by using a screw driver on the actual electronics of the set.

Encapsulation

- **Encapsulation** means hiding the details of an object's internals from the other parts of a program. The object can be used only through its access methods, which are carefully written to keep the object consistent and secure.
- Encapsulation is designed to make an object look like a *black box*: The insides of the box are hidden from view.
- On the outside are some controls which are the only way that the user can use the box.

Interface

- The usual example of this is a TV set where most of the inner workings are sealed off from the user. The user interacts with the set using some well-defined controls. The controls are sometimes called the user **interface**.
- In OO programming, programmers should try to make the interface to the object simple and useful. The inner workings of the object should be made private.

Private Methods

- A **private method** is one that can be used only by the other methods of an object. Parts of a program outside of the object cannot directly invoke (use) a private method of the object.

```
public class CheckingAccount {
  // data-declarations
  private String accountNumber;
  private String accountHolder;
  private int    balance;
  private int    useCount = 0;

  void  processDeposit( int amount ) {
    incrementUse();
    balance = balance + amount ;
  }

  void processCheck( int amount ) {
    int charge;
    incrementUse();
    if ( balance < 100000 )
//     ...
  }
}
```

The public Visibility Modifier

- The private visibility modifier keeps outsiders from looking in. However, the access methods are intended for outsiders, and have to be visible to outsiders in order to be useful.
- The **public** access modifier permits a method or variable of an object to be accessed by code outside of the object.
- The public visibility modifier is used for all access methods and constructors in a class definition. Most variables are made private.

The CheckingAccount Class

```
class CheckingAccount {
  private String accountNumber;
  private String accountHolder;
  private int    balance;
  private int    useCount = 0;

  public CheckingAccount( String accNumber, String holder, int start ) { /*.
     . . .*/ }
  public int currentBalance() { /*. . . .*/ }
  public void  processDeposit( int amount ) { /*. . . .*/ }
  public void processCheck( int amount ) { /*. . . .*/ }
  public void display() { /*. . . . */}

}
```

Package Visibility

- If you do not specify public or private for a variable or a method, then it will have package visibility.
- Default visibility allows a variable or method to be seen within all methods of a class or other classes that are part of the same package.
- A package is a group of related classes.

Parameters

```
public class CheckingAccount  {

// . . . .
    private int balance;
// . . . .
   void processDeposit( int amount ) {
       balance = balance + amount ;
   }
}
```

- The parameter *amount* is used by a caller to send a value to the method. This is usually called *passing a value* to the method.

Parameters (cont'd)

```
public class CheckingAccountTester {

  public static void main( String[] args ) {

    CheckingAccount bobsAccount = new CheckingAccount( "999", "Bob",
     100 );
    bobsAccount.processDeposit( 200 );

    // . . . .
  }
}
```

Parameters (cont'd)

- When the statement bobsAccount.processDeposit(200); is executed, the parameter *amount* of the object's method will hold the value 200. This value is added to the object's instance variable in the statement.
- balance = balance + amount ; Then the method will exit and control will return to *main()*.
- The state of the object referred to by *bobsAccount* will be changed.

Formal and Actual Parameters

- **formal parameter** --- the identifier used in a method to stand for the value that is passed into the method by a caller.
  - For example, *amount* is a formal parameter of *processDeposit*
- **actual parameter** --- the actual value that is passed into the method by a caller.
  - For example, the *200* used when *processDeposit* is called is an actual parameter.
- When a method is called, the formal parameter is temporarily "bound" to the actual parameter. The method can then use a name to stand for the actual value that the caller wanted to be used.

Parameters are Seen by their Own Method, Only

```
public class CheckingAccount {
//  . . . .
  private int    balance;

//  . . . .
  void  processDeposit( int amount ) {
    balance = balance + amount ;
  }

  // modified display method
  void display() {
    System.out.println( balance + "\t" + amount );
  }

}
```

Scope

- The *scope* of a formal parameter (a local variable) is the section of code that can "see" (can use) the parameter.
- The scope of a formal parameter is the body of its method.
- Statements of a method can see the object's instance variables and the object's other methods. They cannot see the parameters (and local variables) of other methods.

```
public class CheckingAccount {
//  . . . .
  private int    balance;
//  . . . .
  void  processDeposit( int amount ) {
                        // scope of amount starts here
    balance = balance + amount ;
  }                     // scope of amount ends here

  void processCheck( int amount ) {
                        // scope of amount starts here
    int charge;
    if ( balance < 100000 )
      charge = 15;
    else
      charge = 0;
    balance =  balance - amount - charge  ;

  }                     // scope of amount ends here

}
```

Assigning to a Parameter

- Within the body of a method, a parameter can be used just like any variable. It can be used in arithmetic expressions, in assignment statements, and so on.
- **However,** for reference by value, changes made to the parameter do not have any effect outside the method body. A parameter is a "local copy" of whatever value the caller passed into the method. Any changes made to it affect only this local copy.

Local Variable

- A **local variable** is a variable that is declared inside of the body of a method. It can be seen only by the statements that follow its declaration inside of that method.
- Its *scope* starts where it is declared and ends at the end of the method.
- Local variables are not used to hold the permanent values of an object's state. They have a value only during the amount of time that a method is active.

Local Variable (cont'd)

```
public class CheckingAccount {
 . . . .
 private int    balance;


 void processCheck( int  amount  ) {
   int charge;   // scope of charge starts here

   if ( balance < 100000 )
     charge = 15;
   else
     charge = 0;

   balance =  balance -  amount  - charge  ;
 }            // scope of charge ends here
}
```

Can't use the Same Name in the Same Scope

```
public class CheckingAccount {
  . . . .
  private int    balance;

  void processCheck( int  amount  ) {
    int amount;                          // It is illegal to use the same
     name
                                         // in the same scope.
    if ( balance < 100000 )
      charge = 15;
    else
      charge = 0;
    balance =  balance -  amount  - amount  ;
  }
}
```

Do Lab#3 Now

Arrays

Array

- An **array** is an object that can be used to store a list of values.
- It is made out of a contiguous block of memory that is divided into a number of "slots."
- Each slot can hold a value, and all of the values are of the same type.

Picture of an Array

	data
0	23
1	38
2	14
3	-3
4	0
5	14
6	9
7	103
8	0
9	-56

- The name of this array is *data*.
- The slots are indexed 0 through 9, and each slot holds an *int*.
- Each slot can be accessed by using its *index*. For example, data[0] is the slot which is indexed by zero (which contains the value 23).

Array

- Indices always start at zero, and count up by one's until the last slot of the array.
- If there are N slots in an array, the indecies will be 0 through N-1.
- Every slot of an array holds a value of the same type.
- A slot of an array that contains a numeric type (such as *int*) can be used anywhere a numeric variable can be used.

Array Declaration

```
type[] arrayName = new type[ length ];
```

- This tells the compiler that *arrayName* will be used as the name of an array containing *type*, and *constructs an array object containing* length number of slots.
- **An array is an object,** and like any other object in Java is constructed out of main storage as the program is running.
- *type*[*length*] names the type of data in each slot and the number of slots. Once an array has been constructed, the number of slots it has does not change.

Example

```
public class arrayEg1 {

  public static void main ( String[] args ) {

    int[] stuff = new int[5];

    stuff[0] = 23;
    stuff[1] = 38;
    stuff[2] = 7*2;

    System.out.println("stuff[0] has " + stuff[0] );
    System.out.println("stuff[1] has " + stuff[1] );
    System.out.println("stuff[2] has " + stuff[2] );
    System.out.println("stuff[3] has " + stuff[3] );
  }
}
```

Output

```
stuff[0] has 23
stuff[1] has 38
stuff[2] has 14
stuff[3] has 0
```

Using a Variable as Index

```
public class arrayEg2 {
  public static void main ( String[] args ) {
    double[] val = new double[4];  //an array of double

    val[0] = 0.12;
    val[1] = 1.43;
    val[2] = 2.98;

    int j = 3;
    System.out.println( val[ j   ] );
    System.out.println( val[ j-1 ] );

    j = j-2;
    System.out.println( val[ j   ] );
  }
}
```

Output

```
0.0
2.98
1.43
```

Initializer List

- You can declare, construct, and initialize the array all in one statement:

```
int[] data = {23, 38, 14, -3, 0, 14, 9, 103, 0, -56 };
```

- This declares an array of *int* which will be named *data*, constructs an *int* array of 10 slots (indexed 0..9), and puts the designated values into the slots. The first value in the *initializer list* corresponds to index 0, the second value coresponds to index 1, and so on.

Two Dimensional Array

- An array with a single subscript is called a *one dimensional* array. The slots of the array can be thought of as lined up one after the other in a line. Two dimensional arrays, three dimensional arrays, and higher dimensional arrays also exist.
- The slots of an array are often called **elements**. In a one dimensional array, the elements are accessed using a single index. In a *two dimensional* array, the elements are accessed using two indices.

Length of an Array

- An array is an object.
- An array object has a member *length* that is the number of elements it has.

```
import java.io.* ;
import java.lang.reflect.Array;

public class inputArray {

  public static void main ( String[] args ) throws IOException {

    int[] array = new int[5];
    int   data;

    BufferedReader inData = new BufferedReader (
                                        new InputStreamReader( System.in ) );

    // input the data
    for ( int index=0; index < array.length; index++ ) {
      System.out.println( "enter an integer: " );
      data          = Integer.parseInt( inData.readLine() );
      array[ index ] = data ;
    }

    // write out the data
    for ( int index=0; index < Array.getLength(array); index++ ) {
      System.out.println( "array[ " + index + " ] = " + array[ index ] );
    }
  }
}
```

Array as a Parameter

- The example programs used a *main()* method which contained the arrays and did all the computing.
- Arrays (and other forms of data) are often manipulated by passing them as *parameters* to specialized methods.
- Each method is like a "tool" that does something with an array.

Array as a Parameter (cont'd)

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

  void print( int[ ] x ) {
    for ( int index=0; index < x.length; index++ )
      System.out.print( x[index] + " " );
    System.out.println();
  }
}
class ArrayDemo {

  public static void main ( String[ ] args ) {
    ArrayOps operate = new ArrayOps();
    int[] ar1 =  { -20, 19, 1, 5, -1, 27, 19, 5 } ;
    System.out.print  ("\nThe array is: " );
    operate.print( ar1 );
  }
}
```

Array as a Parameter (cont'd)

1. It defines a class called *ArrayOps*.

- *ArrayOps* contains a method *print()* to print out each element of an array.
- The parameter list is: *int[] x*
- This declares a formal parameter *x* which will be a reference to an array object.
- The caller is expected to supply a reference to an array as an actual parameter

Array as a Parameter (cont'd)

2. It defines a class called *ArrayDemo*.
- *ArrayDemo* holds the static *main()* class.
- The static *main()* class creates an array object.
- The static *main()* class creates an *ArrayOps* object, *ar1*.
- The *print()* method of the *ArrayOps* object is "called" with *a reference* to the array object *ar1* as a parameter.

Fixed-length Array

- Once an array object has been constructed, the number of slots will not change.
- If an array object is to be used for the entire run of a program, its length must be large enough to accommodate all expected data.
- It is often hard to guess the correct length in advance.

Vectors

Vector *Class*

- The *Vector* class builds upon the capabilities of arrays.
- A *Vector* object contains an array of object references plus many methods for managing that array.
- You can keep adding elements to it no matter what the original size of the vector is.
- The size of the *Vector* will increase automatically and no information will be lost.

Vector *Class (cont'd)*

- The elements of a *Vector* are *object references*, not primitive data.
- Using a *Vector* is slightly slower than using an array.
- The elements of a *Vector* are references to *Object*. This means that often you will need to use type casting with the data from a *Vector*.

Constructors of Vector *Objects*

```
Vector myVector;                          // myVector is a reference to a future
                                          // vector object

Vector myVector = new Vector();           // myVector is a reference to a Vector
object.
                                          // The Java system picks the initial
capacity.

Vector myVector = new Vector(15);         // myVector is a reference to a Vector
object
                                                // with an initial capacity of 15
elements.

Vector myVector = new Vector(15, 5);      // myVector is a reference to a Vector
object
                                                // with an initial capacity of 15
elements,
                                                // and an increment size of 5
```

Capacity and Size

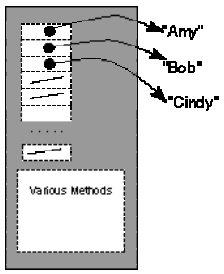
- The *capacity* is the number of slots available.
- The *size* is the number of slots that have data in them.
- Slots 0 up through size-1 have data in them.
- You cannot skip slots; before slot N gets data, slots, 0, 1, 2, ... N-1 must be filled with data.

Adding Elements to a Vector

```
import java.util.* ;

class VectorEg {

  public static void main ( String[] args) {
    Vector names = new Vector( 20, 5 );

    System.out.println("capacity: " + names.capacity() );
    System.out.println("size: " + names.size() );

    names.addElement("Amy");
    names.addElement("Bob");
    names.addElement("Cindy");

    System.out.println("capacity: " + names.capacity() );
    System.out.println("size: " + names.size() );
  }
}
```

Results

```
capacity: 20
size: 0
capacity: 20
size: 3
```



After three addElement()

Printing all Elements in a Vector

```
import java.util.* ;

class VectorEg {

  public static void main ( String[] args) {
    Vector names = new Vector( 10 );

    names.addElement( "Amy" );
    names.addElement( "Bob" );
    names.addElement( "Chris" );
    names.addElement( "Dan" );
    names.addElement( "Elaine" );
    names.addElement( "Fred" );

    for ( int j=0; j < name size() ___; j++ )
      System.out.println( j + ": " + nam elementAt ____(j) );
  }
}
```

Removing an Element

```
removeElementAt(int index)  //  Deletes the element at index. Each element
with an
                            //  index greater than index is shifted downward to
have                        //an index one smaller than its previous value.
```

- The element at location index will be eliminated. Elements at locations index+1, index+2, ... , size()-1 will each be moved down one to fill in the gap.
- This is like pulling out a book from the middle of a stack of books.

Example

```
import java.util.* ;
class VectorEg {

  public static void main ( String[] args) {
    Vector names = new Vector( 10 );

    names.addElement( "Amy" );
    names.addElement( "Bob" );
    names.addElement( "Chris" );
    names.addElement( "Deb" );

    names.removeElementAt(2);

    for ( int j=0; j < names.size(); j++ )
      System.out.println( j + ": " + names.elementAt(j) );
  }
}
```

Results

```
0: Amy
1: Bob
2: Deb
```

Inserting Elements

```
insertElementAt(Object element, int index)  // Inserts the element at
index.
                                            // Each element with an
index
                                            // equal or greater than index
is
                                            // shifted upward by one more
than
                                            // its previous value.
```

Example

```
import java.util.* ;
class VectorEg {

  public static void main ( String[] args) {
    Vector names = new Vector( 10 );

    names.addElement( "Amy" );
    names.addElement( "Bob" );
    names.addElement( "Chris" );
    names.addElement( "Deb" );

    names.insertElementAt( "Elaine", 2);

    for ( int j=0; j < names.size(); j++ )
      System.out.println( j + ": " + names.elementAt(j)
);

  }
}
```

```
0: Amy
1: Bob
2: Elaine
3: Chris
4: Deb
```

FirstElement() *and* lastElement()

```
import java.util.* ;
class VectorEg {

  public static void main ( String[] args) {
    Vector names = new Vector( 10 );

    names.addElement( "Amy" );
    names.addElement( "Bob" );
    names.addElement( "Chris" );
    names.addElement( "Deb" );

    System.out.println( names.firstElement() );
    System.out.println( names.lastElement() );

  }
}
```

Amy
Deb

isEmpty()

- The *elementAt()*, *removeElementAt()*, and other methods will also throw an exception when the designated element does not exist.
- To check if a Vector has elements use boolean *isEmpty()* which returns *true* if the Vector has no elements.

Inheritance

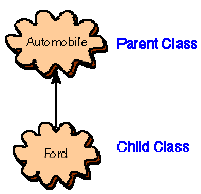
Inheritance

- Inheritance enables you to define a new class based on a class that already exists.
- The new class will be similar to the existing class, but will have some new characteristics.

Simple Single Inheritance

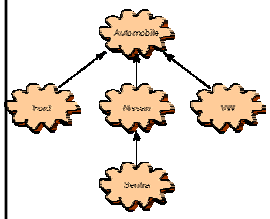
- The class that is used as a basis for defining a new class is called a *parent* class (or superclass or base class.)
- The new class based on the parent class is called a *child* class (or subclass or derived class.)
- In Java, children inherit characteristics from *just one* parent. This is called *single inheritance*.

Is-a Relationship



- The picture shows a parent class and a child class. The line between them shows the "is-a" relationship.
- Inheritance is between *classes*, not between objects.

Hierarchies



- In a hierarchy, each class has at most one parent, but might have several child classes.
- There is one class, at the "top" of the hierarchy that has no parent. This is sometimes called the **root** of the hierarchy.

Syntax of Deriving a Child Class

```
public class childClass extends parentClass {

  // new characteristics of the child class go here

}
```

Tape Store Example

```
public class VideoTape {

  String  title;                    // name of the item
  int     length;                   // number of minutes
  boolean avail;                    // is the tape in the store?

  public VideoTape( String ttl, int lngth ) {
    title = ttl; length = lngth; avail = true;
  }

  public void show() {
    System.out.println( title + ", " + length + " min. available:" + avail );
  }
 }

public class TapeStore {

  public static void main ( String args[] ) {
    VideoTape item1 = new VideoTape("Jaws", 120 );   item1.show();
  }
}
```

```
public class VideoTape {

  String  title;                    // name of the item
  int     length;                   // number of minutes
  boolean avail;                    // is the tape in the store?

  public VideoTape( String ttl, int lngth ) {

    title = ttl; length = lngth; avail = true;
  }
  public void show() {

    System.out.println( title + ", " + length + " min. available:" + avail );
  }
}
public class Movie extends VideoTape {

  String  director;                 // name of the director
  String  rating;                   // G, PG, R, or X

  public Movie( String ttl, int lngth, String dir, String rtng ) {

    super( ttl, lngth );            // use the super class' constuctor
    director = dir;  rating = rtng;     // initialize what's new to Movie
  }
}
```

Using Inheritance

- The class *Movie* is a subclass of *VideoTape*. An object of type *Movie* has the following members in it:
  - title inherited from VideoTape
  - length inherited from VideoTape
  - avail inherited from VideoTape
  - show() inherited from VideoTape
  - director defined in Movie
  - rating defined in Movie

Using a Super Class's Constructor

```
// constructor
 public Movie( String ttl, int lngth, String dir, String rtng ) {
   super( ttl, lngth );             // use the super class' constuctor
   director = dir;  rating = rtng;     // initialize what's new to Movie
 }
```

- The statement *super(ttl, lngth)* invokes the super class' constructor to initialize some of the data. Then the next two statements initialize the members that only *Movie* has.
- When *super* is used in this way, it must be the first statement in the subclass' constructor.

Instantiating Movie

```
public class TapeStore {

  public static void main ( String args[] ) {
    VideoTape item1 = new VideoTape("Microcosmos", 90 );
    Movie     item2 = new Movie("Jaws", 120, "Spielberg", "PG" );
    item1.show();
    item2.show();
  }
}
```

Output:

Microcosmos, 90 min. available:true

Jaws, 120 min. available:true

Overriding Methods

```
// added to class Movie
 public void show() {
   System.out.println( title + ", " + length + " min. available:" + avail );
   System.out.println( "dir: " + director + "  " + rating );
 }
```

- A child *overrides* a method from its parent by defining a replacement method with the same signature.
- The parent has its method, and the child has its own method with the same name.

Overriding Methods (cont'd)

```
public class TapeStore{

  public static void main ( String args[] ) {
    VideoTape item1 = new VideoTape("Microcosmos", 90 );
    Movie     item2 = new Movie("Jaws", 120, "Spielberg", "PG" );
    item1.show();
    item2.show();
  }
}
```

Output:

Microcosmos, 90 min. available:true

Jaws, 120 min. available:true

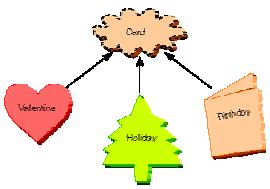
dir: Spielberg PG

Using super in Child's Methods

```
public void show() {
   super.show();
   System.out.println( "dir: " + director + "  " + rating );
}
```

- Unlike the case when *super* is used in a constructor, inside a method *super* does not have to be used in the first statement.

Greeting Card Hierarchy



- The parent class is *Card*, and its children classes are *Valentine*, *Holiday*, and *Birthday*.
- An object must be an instance of one of the three child types: *Valentine*, *Holiday*, and *Birthday*.
- There will be no such thing as an object that is merely a "Card."

Abstract Classes

Abstract Class

- An **abstract class** in Java is a class that is never instantiated. Its purpose is to be a parent to several related classes. The child classes will inherit from the abstract parent class.
- An abstract class is defined like this:

```
abstract class ClassName {

  . . . . . // definitions of methods and variables

}
```

Abstract Methods

- Each card class will have its own version of the *greeting()* method, but each one is implemented differently, it is useful to put an abstract *greeting()* method in the parent class.

```
abstract class Card {
  String recipient;                    // name of who gets the card
  public abstract void greeting();  // abstract greeting() method
}
```

- Abstract classes can (but don't have to) contain abstract methods. Also, an abstract class can contain non-abstract methods, which will be inherited by the children.

Abstract Methods (cont'd)

- An **abstract method** has no body.
- It declares an access modifier, return type, and method signature followed by a semicolon.
- A non-abstract child class inherits the abstract method and must define all abstract methods that match the abstract method.
- An *abstract* child of an abstract parent does not have to define non-abstract methods for the abstract signatures it inherits.

Holiday

```
class Holiday extends Card {

  public Holiday( String r ) {
    recipient = r;
  }

  public void greeting() {
    System.out.println("Dear " + recipient + ",\n");
    System.out.println("Season's Greetings!\n\n");
  }
}
```

Holiday (cont'd)

- The class Holiday is not an abstract class; objects can be instantiated from it.
- *Holiday* inherites the abstract method *greeting()* from its parent.
- *Holiday* must define a *greeting()* method that includes a method body.
- The definition of *greeting()* must match the signature given in the parent.
- If *Holiday* did not define *greeting()*, then *Holiday* would have to be declared to be an abstract class.

Abstract Class

- Not everything defined in an abstract class needs to be abstract.
- However, if a class contains even one abstract method, then the class itself must be declared to be abstract.

```
abstract class Card {
  String recipient;
  public abstract void greeting();
}
class Holiday extends Card {

  public Holiday( String r ) {
    recipient = r;
  }

  public void greeting() {
    System.out.println("Dear " + recipient + ",\n");
    System.out.println("Season's Greetings!\n\n");
  }
}

public class CardTester {

  public static void main ( String[] args ) {
    Holiday hol = new Holiday("Santa");
    hol.greeting();
  }
}
```

Advantage of Abstract Class

- You can group several related classes together as siblings, even though none is the parent of the others.
- Grouping classes together is important in keeping a program organized and understandable.

Birthday Class

```
class Birthday exten  Card  ___ {

  int age;

  public  Birthday  __ ( String r, int years ) {
    recipient = r;
    age = years;
  }

  public void greeting() {
    System.out.println("Dear " + recipient + ",\n");
    System.out.println("Happy "  age  __ + "th Birthday\n\n");
  }
}
```

Valentine Class

```
class Valentine extends Card {

 int kisses;

 public Valentine ( String r, int k ) {
  recipient = r;
  kisses    = k;
 }

 public void greeting() {
  System.out.println("Dear " + recipient + ",\n");
  System.out.println("Love and Kisses,\n");
  for ( int j=0; j<kisses; j++ )
   System.out.print("X");
  System.out.println("\n\n");
 }
}
```

Complete Program

```
import java.io.*;

abstract class Card {
 String recipient;
 public abstract void greeting();
}
```

Complete Program (cont'd)

```
class Holiday extends Card {

 public Holiday( String r ) {
  recipient = r;
 }

 public void greeting() {
  System.out.println("Dear " + recipient + ",\n");
  System.out.println("Season's Greetings!\n\n");
 }
}
```

Complete Program (cont'd)

```
class Birthday extends Card {

  int age;

  public Birthday ( String r, int years ) {
    recipient = r;
    age = years;
  }

  public void greeting() {
    System.out.println("Dear " + recipient + ",\n");
    System.out.println("Happy " + age + "th Birthday\n\n");
  }
}
```

Complete Program (cont'd)

```
class Valentine extends Card {

  int kisses;

  public Valentine ( String r, int k ) {
    recipient = r;
    kisses    = k;
  }

  public void greeting() {
    System.out.println("Dear " + recipient + ",\n");
    System.out.println("Love and Kisses,\n");
    for ( int j=0; j < kisses; j++ )
      System.out.print("X");
    System.out.println("\n\n");
  }
}
```

Complete Program (cont'd)

```
public class CardTester {

  public static void main ( String[] args ) throws IOException {
    String me;
    BufferedReader input = new BufferedReader(
                                        new
InputStreamReader(System.in) );
    System.out.println("Your name:");
    me = input.readLine();

    Holiday   hol = new Holiday( me );
    hol.greeting();

    Birthday  bd  = new Birthday( me, 21 );
    bd.greeting();

    Valentine val = new Valentine( me, 7 );
    val.greeting();
  }
```

A Sample Run

```
Your name:
Sue
Dear Sue,

Season's Greetings!

Dear Sue,

Happy 21th Birthday

Dear Sue,

Love and Kisses,

XXXXXXX
```

Can't Instantiate an Abstract Class

```
. . . .
public static void main ( String[] args ) throws IOException {
  . . . .
  Card card = new Card() ;       // can't instantiate abstract class
  card.greeting() ;
  . . . .
}
```

This is OK

```
. . . .
public static void main ( String[] args ) throws IOException {
  . . . .
  Card card = new Valentine( "Joe", 14 ) ;     // a Valentine is-a Card
  card.greeting() ;
  . . . .
}
```

Class Polymorphism

Using Parent Class Reference Variables

- A reference variable of class "C" can be used with any object that is related by inheritance to class "C".
- Usually, references to a parent class are used to hold children objects.
- When a method is invoked, it is the class of the object (not of the variable) that determines which method is run.

Polymorphism

- **Polymorphism** means "having many forms."
- In Java, it means that a single variable might be used with several different types of related objects at different times in a program.
- When the variable is used with "dot notation" to invoke a method, exactly which method is run depends on the object that the variable currently refers to.

Example

```
. . . .                    // class definitions as before

public class CardTester {

  public static void main ( String[] args ) throws IOException {

    Card card = new Holiday( "Amy" );
    card.greeting();              //Invoke a Holiday greeting()

    card = new Valentine( "Bob", 3 );
    card.greeting();              //Invoke a Valentine greeting()

    card = new Birthday( "Cindy", 17 );
    card.greeting();              //Invoke a Birthday greeting()

  }
}
```

What types of objects a reference variable can refer to

- A variable can hold a reference to an object who's class is a descendant of the class of the variable.
- The class of the object must be *a descendant* of the class of the variable that holds a reference to that object.
- *A descendant* of a class is a child of that class, or a child of a child of that class, and so on. Siblings are not descendants of each other

Do Lab#4 Now

(If you are not tired yet…)

Reference

- [Lewis01]John Lewis, William Loftus, Java Software Solutions, Addison Wesley, 2001.
- [Kjell00]Bradley Kjell, Introduction to Computer Science using Java.
- [Mancorids01]Spiros Mancorids, Software Design class notes. http://www.mcs.drexel.edu/~smancori/teaching/CS575/slides/java.ppt