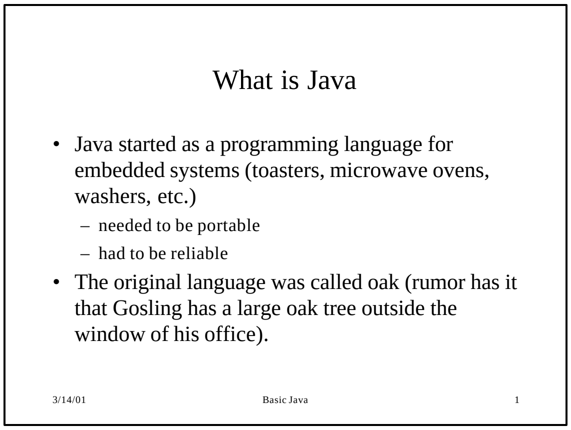
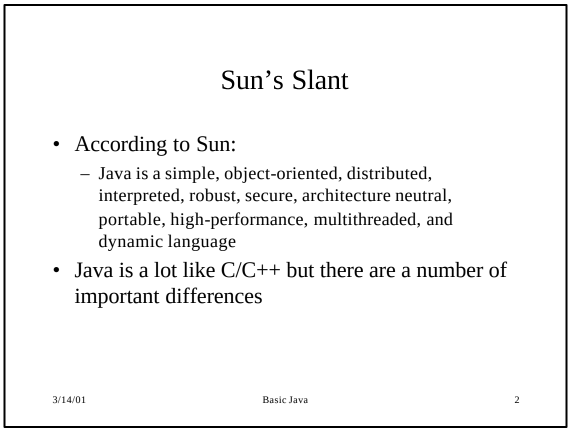


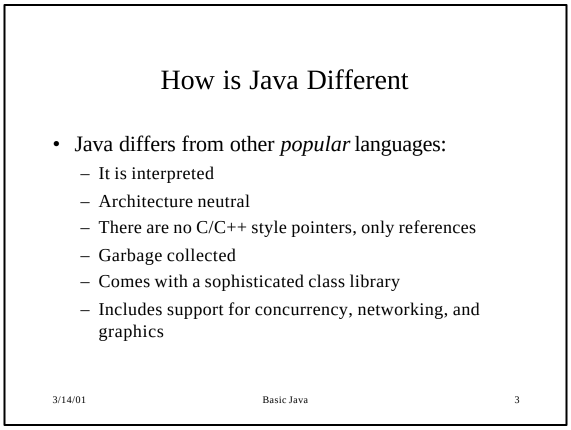
What is Java

- Java started as a programming language for embedded systems (toasters, microwave ovens, washers, etc.)
  - needed to be portable
  - had to be reliable
- The original language was called oak (rumor has it that Gosling has a large oak tree outside the window of his office).



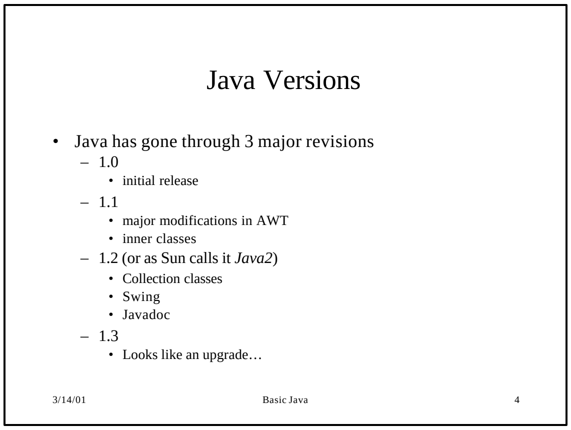
Sun’s Slant

- According to Sun:
  - Java is a simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language
- Java is a lot like C/C++ but there are a number of important differences



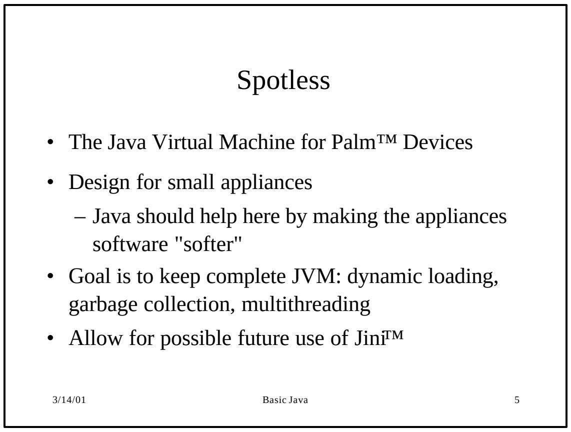
How is Java Different

- Java differs from other *popular* languages:
  - It is interpreted
  - Architecture neutral
  - There are no C/C++ style pointers, only references
  - Garbage collected
  - Comes with a sophisticated class library
  - Includes support for concurrency, networking, and graphics



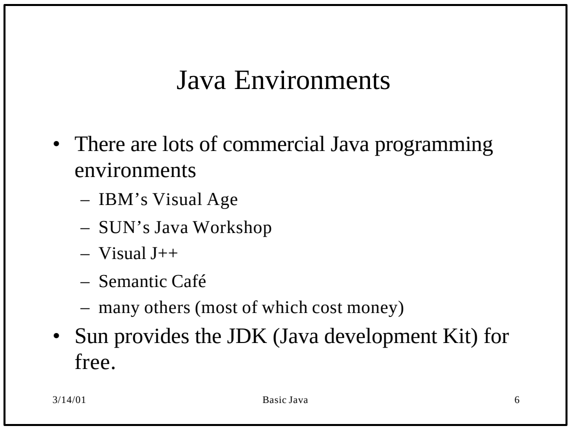
Java Versions

- Java has gone through 3 major revisions
  - 1.0
    - initial release
  - 1.1
    - major modifications in AWT
    - inner classes
  - 1.2 (or as Sun calls it *Java2*)
    - Collection classes
    - Swing
    - Javadoc
  - 1.3
    - Looks like an upgrade…



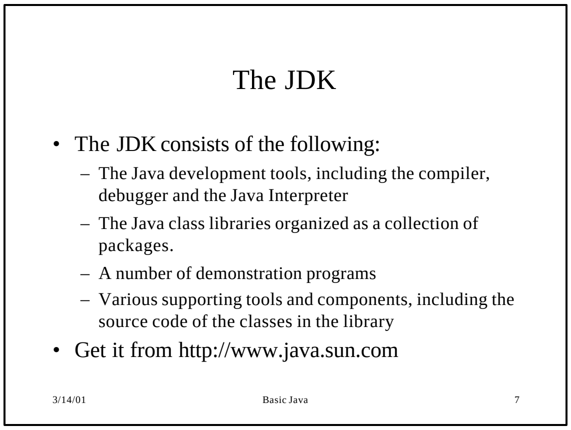
Spotless

- The Java Virtual Machine for Palm™ Devices
- Design for small appliances
  - Java should help here by making the appliances software "softer"
- Goal is to keep complete JVM: dynamic loading, garbage collection, multithreading
- Allow for possible future use of Jini™



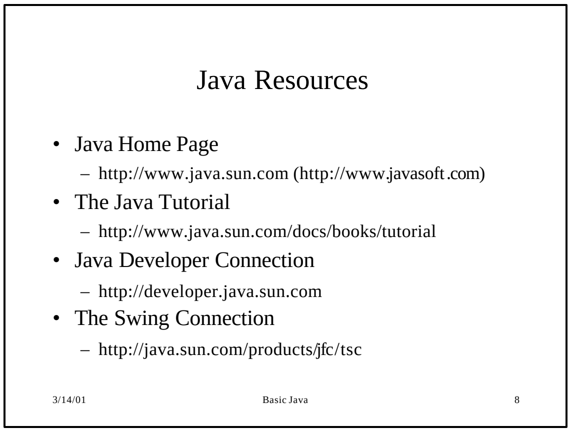
Java Environments

- There are lots of commercial Java programming environments
  - IBM’s Visual Age
  - SUN’s Java Workshop
  - Visual J++
  - Semantic Café
  - many others (most of which cost money)
- Sun provides the JDK (Java development Kit) for free.



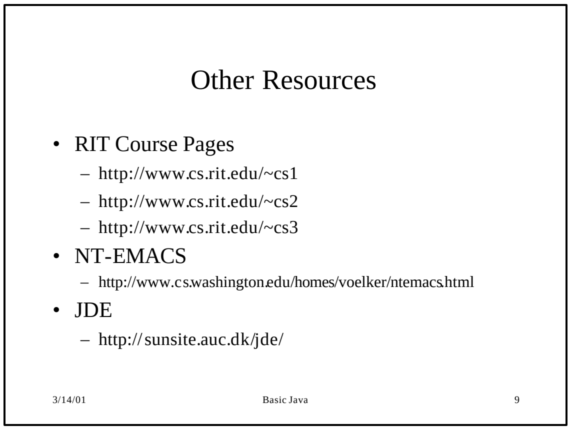
The JDK

- The JDK consists of the following:
  - The Java development tools, including the compiler, debugger and the Java Interpreter
  - The Java class libraries organized as a collection of packages.
  - A number of demonstration programs
  - Various supporting tools and components, including the source code of the classes in the library
- Get it from http://www.java.sun.com



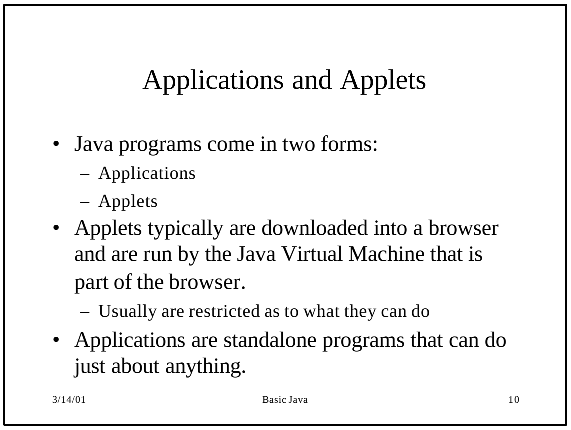
Java Resources

- Java Home Page
  - http://www.java.sun.com (http://www.javasoft.com)
- The Java Tutorial
  - http://www.java.sun.com/docs/books/tutorial
- Java Developer Connection
  - http://developer.java.sun.com
- The Swing Connection
  - http://java.sun.com/products/jfc/tsc



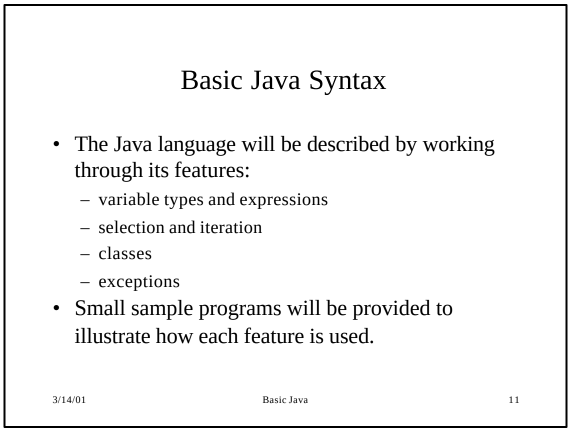
Other Resources

- RIT Course Pages
  - http://www.cs.rit.edu/~cs1
  - http://www.cs.rit.edu/~cs2
  - http://www.cs.rit.edu/~cs3
- NT-EMACS
  - http://www.cs.washington.edu/homes/voelker/ntemacs.html
- JDE
  - http://sunsite.auc.dk/jde/



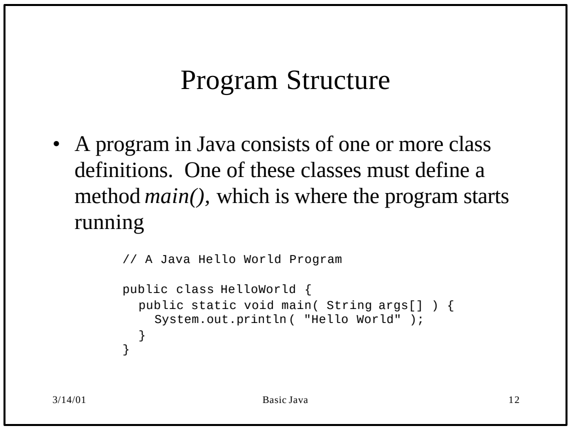
Applications and Applets

- Java programs come in two forms:
  - Applications
  - Applets
- Applets typically are downloaded into a browser and are run by the Java Virtual Machine that is part of the browser.
  - Usually are restricted as to what they can do
- Applications are standalone programs that can do just about anything.



Basic Java Syntax

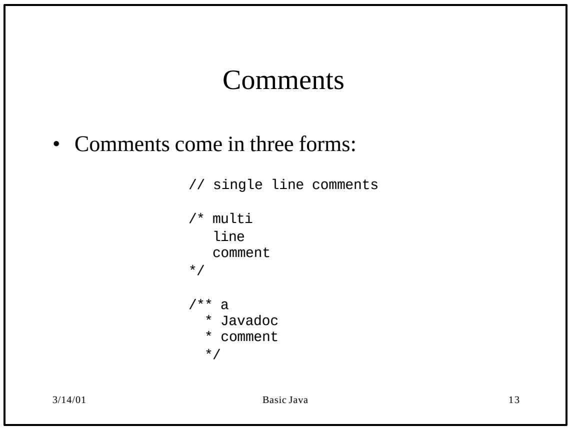
- The Java language will be described by working through its features:
  - variable types and expressions
  - selection and iteration
  - classes
  - exceptions
- Small sample programs will be provided to illustrate how each feature is used.



Program Structure

- A program in Java consists of one or more class definitions. One of these classes must define a method *main()*, which is where the program starts running

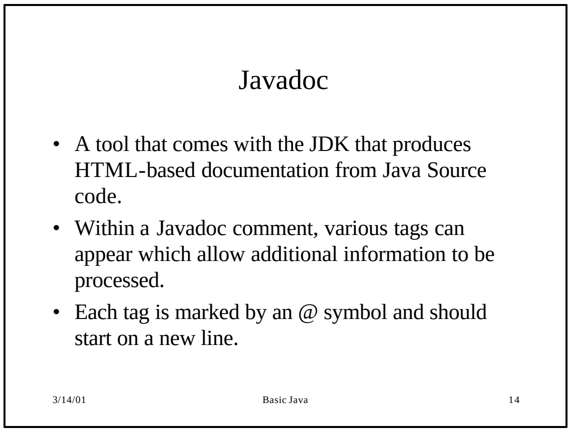
```
// A Java Hello World Program

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



Comments

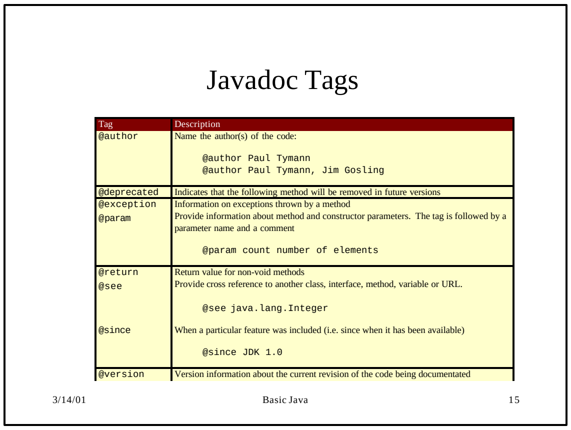
- Comments come in three forms:

```
// single line comments

/* multi
   line
   comment
*/

/** a
  * Javadoc
  * comment
  */
```



Javadoc

- A tool that comes with the JDK that produces HTML-based documentation from Java Source code.
- Within a Javadoc comment, various tags can appear which allow additional information to be processed.
- Each tag is marked by an @ symbol and should start on a new line.

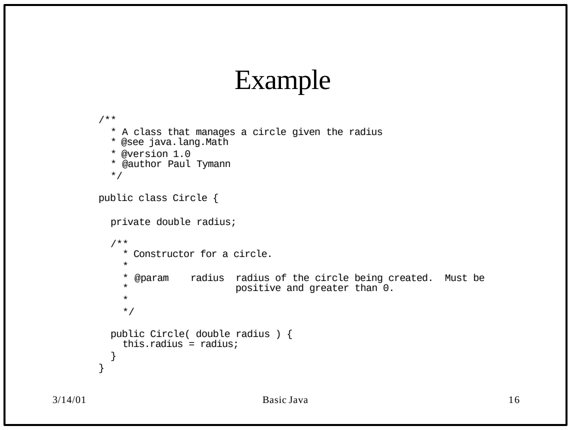


Javadoc Tags

Tag	Description

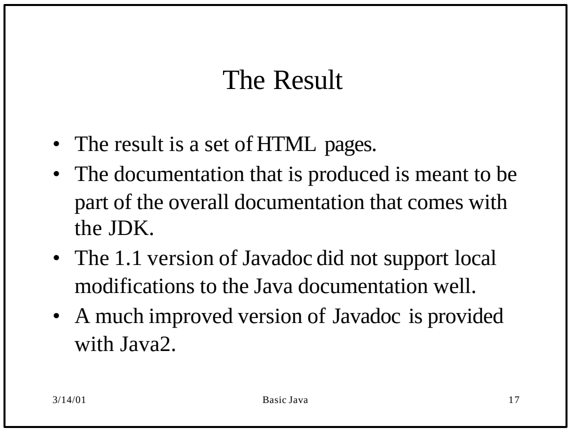
`@author`	Name the author(s) of the code:
	`@author Paul Tymann`
	`@author Paul Tymann, Jim Gosling`

`@deprecated`	Indicates that the following method will be removed in future versions
`@exception`	Information on exceptions thrown by a method
`@param`	Provide information about method and constructor parameters. The tag is followed by a parameter name and a comment `@param count number of elements`
`@return`	Return value for non-void methods
`@see`	Provide cross reference to another class, interface, method, variable or URL. `@see java.lang.Integer`
`@since`	When a particular feature was included (i.e. since when it has been available) `@since JDK 1.0`
`@version`	Version information about the current revision of the code being documentated



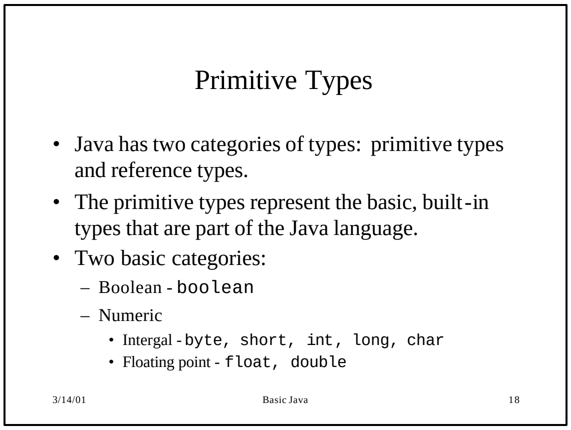
Example

```
/**
  * A class that manages a circle given the radius
  * @see java.lang.Math
  * @version 1.0
  * @author Paul Tymann
  */

public class Circle {

  private double radius;

  /**
    * Constructor for a circle.
    *
    * @param    radius  radius of the circle being created.  Must be
    *                   positive and greater than 0.
    *
    */

  public Circle( double radius ) {
    this.radius = radius;
  }
}
```



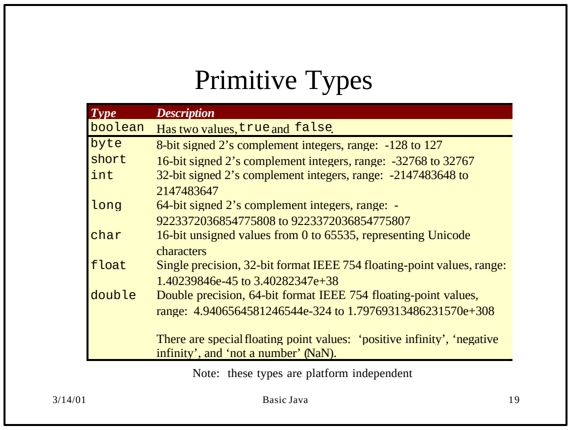
The Result

- The result is a set of HTML pages.
- The documentation that is produced is meant to be part of the overall documentation that comes with the JDK.
- The 1.1 version of Javadoc did not support local modifications to the Java documentation well.
- A much improved version of Javadoc is provided with Java2.



Primitive Types

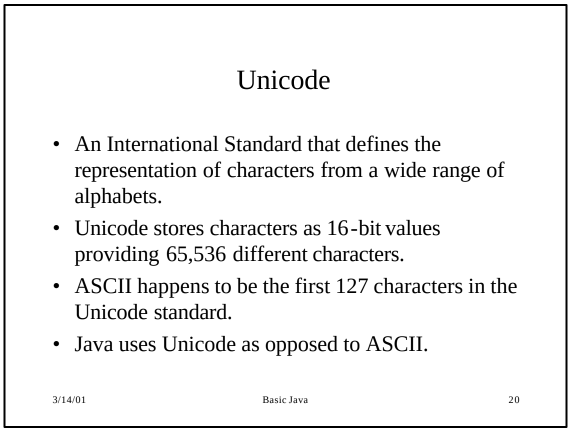
- Java has two categories of types: primitive types and reference types.
- The primitive types represent the basic, built-in types that are part of the Java language.
- Two basic categories:
  - Boolean - `boolean`
  - Numeric
    - Intergal - `byte, short, int, long, char`
    - Floating point - `float, double`



Primitive Types

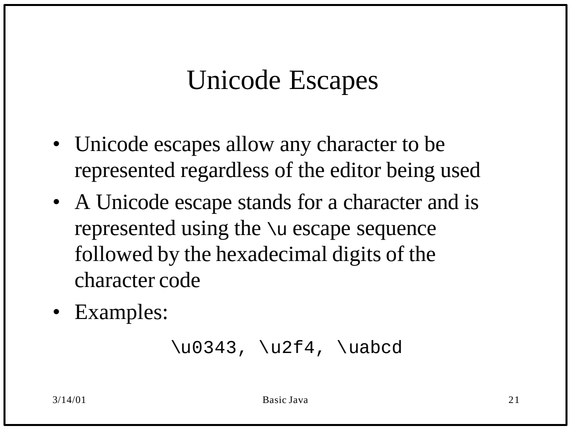
Type	*Description*
`boolean`	Has two values, `true` and `false`.
`byte`	8-bit signed 2’s complement integers, range: -128 to 127
`short`	16-bit signed 2’s complement integers, range: -32768 to 32767
`int`	32-bit signed 2’s complement integers, range: -2147483648 to 2147483647
`long`	64-bit signed 2’s complement integers, range: -9223372036854775808 to 9223372036854775807
`char`	16-bit unsigned values from 0 to 65535, representing Unicode characters
`float`	Single precision, 32-bit format IEEE 754 floating-point values, range: 1.40239846e-45 to 3.40282347e+38
`double`	Double precision, 64-bit format IEEE 754 floating-point values, range: 4.9406564581246544e-324 to 1.79769313486231570e+308 There are special floating point values: ‘positive infinity’, ‘negative infinity’, and ‘not a number’ (NaN).

Note: these types are platform independent



Unicode

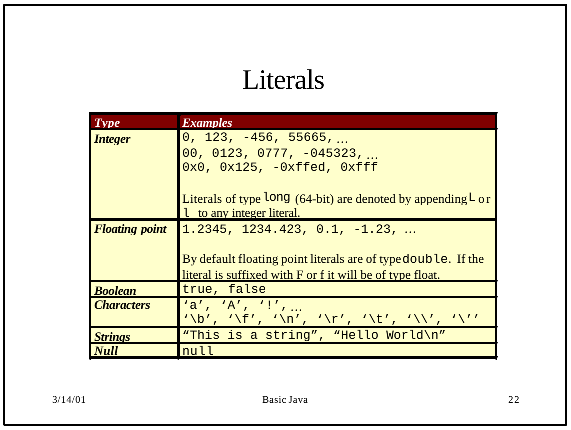
- An International Standard that defines the representation of characters from a wide range of alphabets.
- Unicode stores characters as 16-bit values providing 65,536 different characters.
- ASCII happens to be the first 127 characters in the Unicode standard.
- Java uses Unicode as opposed to ASCII.



Unicode Escapes

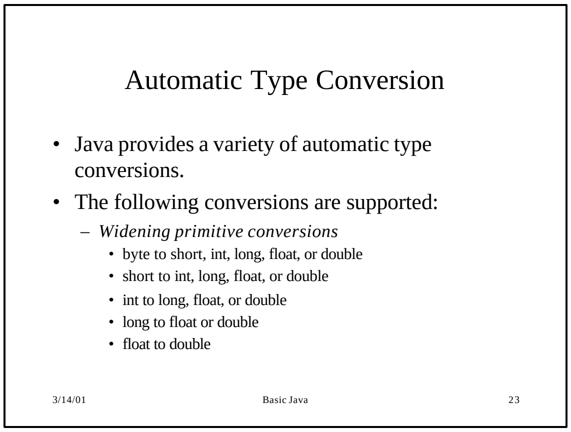
- Unicode escapes allow any character to be represented regardless of the editor being used
- A Unicode escape stands for a character and is represented using the `\u` escape sequence followed by the hexadecimal digits of the character code
- Examples:

```
\u0343, \u2f4, \uabcd
```



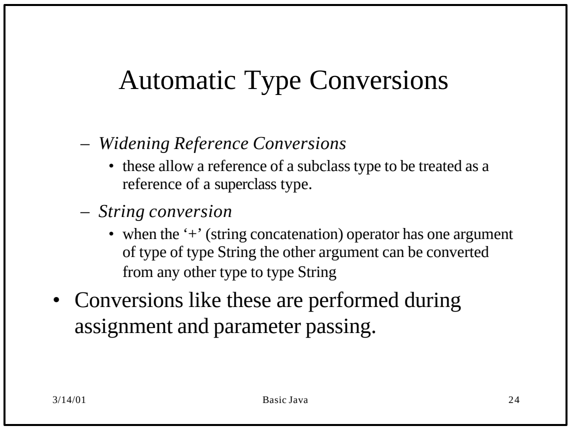
Literals

Type	*Examples*
Integer	`0, 123, -456, 55665,` … `00, 0123, 0777, -045323,` … `0x0, 0x125, -0xffed, 0xfff` Literals of type `long` (64-bit) are denoted by appending `L` or `l` to any integer literal.
Floating point	`1.2345, 1234.423, 0.1, -1.23,` … By default floating point literals are of type `double`. If the literal is suffixed with F or f it will be of type float.
Boolean	`true, false`
Characters	`‘a’, ‘A’, ‘!’,` … `‘\b’, ‘\f’, ‘\n’, ‘\r’, ‘\t’, ‘\\’, ‘\’’`
Strings	`“This is a string”, “Hello World\n”`
Null	`null`



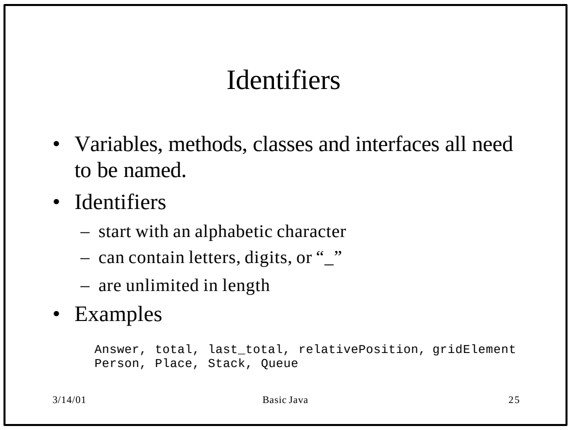
Automatic Type Conversion

- Java provides a variety of automatic type conversions.
- The following conversions are supported:
  - – *Widening primitive conversions*
    - byte to short, int, long, float, or double
    - short to int, long, float, or double
    - int to long, float, or double
    - long to float or double
    - float to double



Automatic Type Conversions

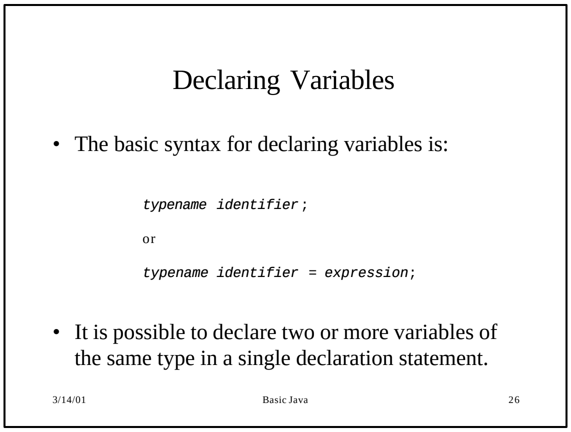
- – *Widening Reference Conversions*
  - these allow a reference of a subclass type to be treated as a reference of a superclass type.
- – *String conversion*
  - when the ‘+’ (string concatenation) operator has one argument of type of type String the other argument can be converted from any other type to type String
- Conversions like these are performed during assignment and parameter passing.



Identifiers

- Variables, methods, classes and interfaces all need to be named.
- Identifiers
  - start with an alphabetic character
  - can contain letters, digits, or "_"
  - are unlimited in length
- Examples

```
Answer, total, last_total, relativePosition, gridElement
Person, Place, Stack, Queue
```



Declaring Variables

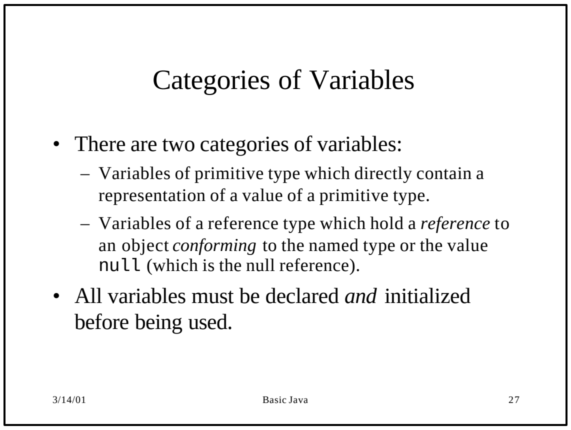
- The basic syntax for declaring variables is:

```
typename identifier;
```

or

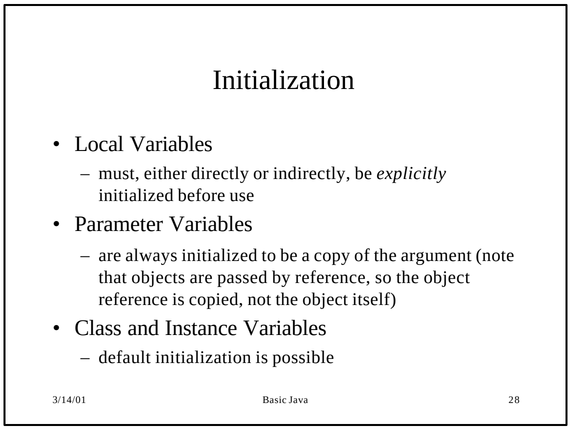
```
typename identifier = expression;
```

- It is possible to declare two or more variables of the same type in a single declaration statement.



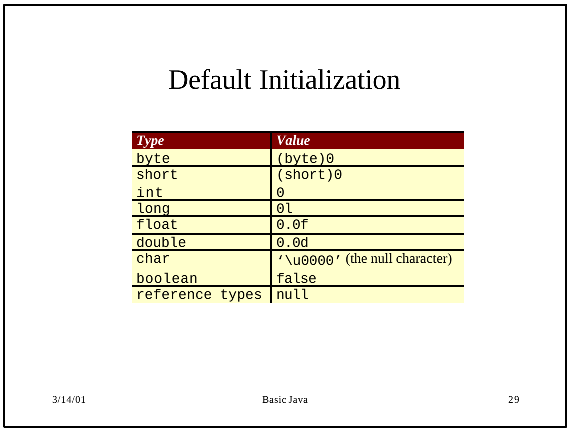
Categories of Variables

- There are two categories of variables:
  - Variables of primitive type which directly contain a representation of a value of a primitive type.
  - Variables of a reference type which hold a *reference* to an object *conforming* to the named type or the value `null` (which is the null reference).
- All variables must be declared *and* initialized before being used.



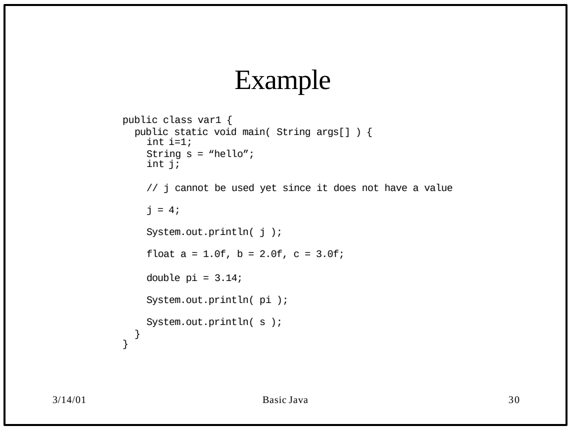
Initialization

- Local Variables
  - must, either directly or indirectly, be *explicitly* initialized before use
- Parameter Variables
  - are always initialized to be a copy of the argument (note that objects are passed by reference, so the object reference is copied, not the object itself)
- Class and Instance Variables
  - default initialization is possible



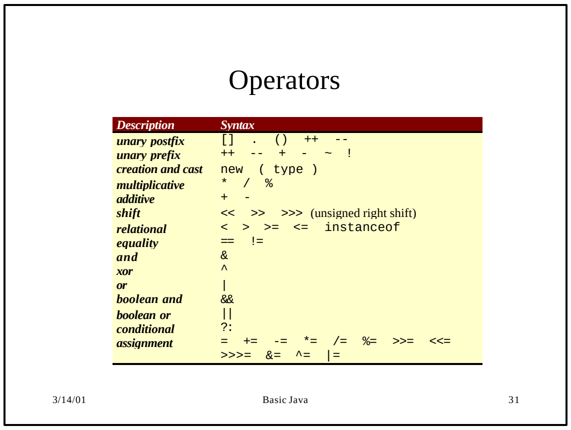
Default Initialization

Type	***Value***
`byte`	`(byte)0`
`short`	`(short)0`
`int`	`0`
`long`	`0l`
`float`	`0.0f`
`double`	`0.0d`
`char`	`‘\u0000’` (the null character)
`boolean`	`false`
`reference types`	`null`



Example

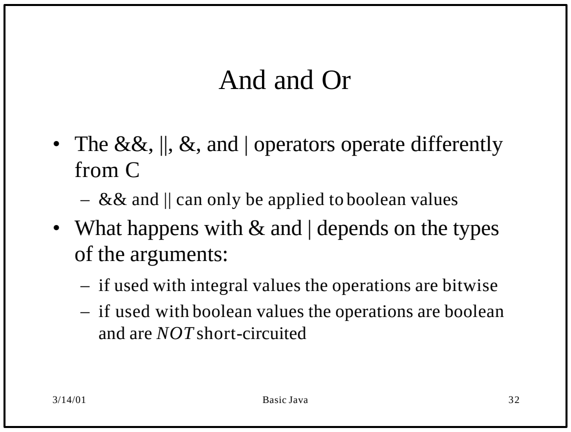
```
public class var1 {
  public static void main( String args[] ) {
    int i=1;
    String s = “hello”;
    int j;

    // j cannot be used yet since it does not have a value

    j = 4;

    System.out.println( j );

    float a = 1.0f, b = 2.0f, c = 3.0f;

    double pi = 3.14;

    System.out.println( pi );

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



Operators

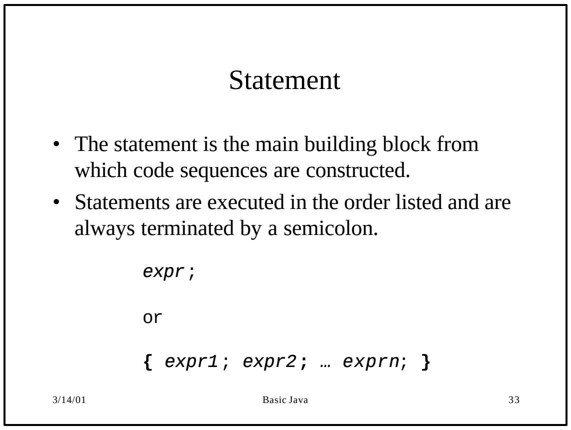
Description	*Syntax*
unary postfix	`[]   .   ()   ++   --`
unary prefix	`++   --   +   -   ~   !`
creation and cast	`new  ( type )`
multiplicative	`*   /   %`
additive	`+   -`
shift	`<<   >>   >>>` (unsigned right shift)
relational	`<   >   >=   <=   instanceof`
equality	`==   !=`
and	`&`
xor	`^`
or	`|`
boolean and	`&&`
boolean or	`||`
conditional	`?:`

assignment	`=   +=   -=   *=   /=   %=   >>=   <<=`
	`>>>=   &=   ^=   |=`



And and Or

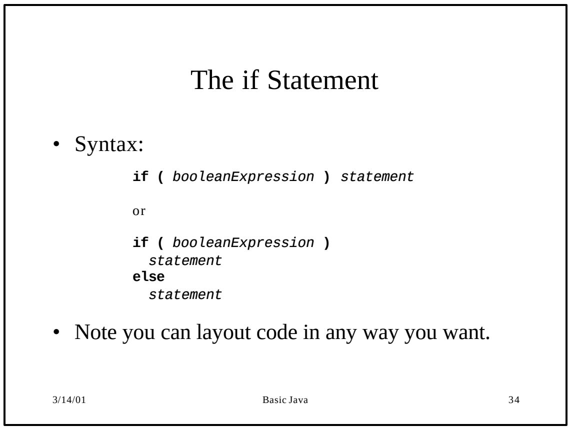
- The &&, ||, &, and | operators operate differently from C
  - && and || can only be applied to boolean values
- What happens with & and | depends on the types of the arguments:
  - if used with integral values the operations are bitwise
  - if used with boolean values the operations are boolean and are *NOT* short-circuited



Statement

- The statement is the main building block from which code sequences are constructed.
- Statements are executed in the order listed and are always terminated by a semicolon.

```
expr;

or

{ expr1; expr2; … exprn; }
```

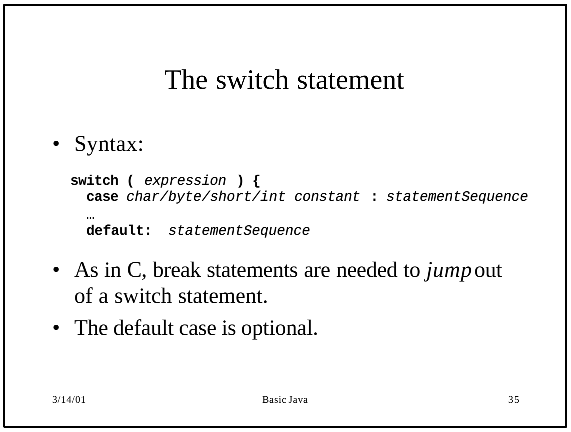


The if Statement

- Syntax:

```
if ( booleanExpression ) statement

or

if ( booleanExpression )
  statement
else
  statement
```

- Note you can layout code in any way you want.

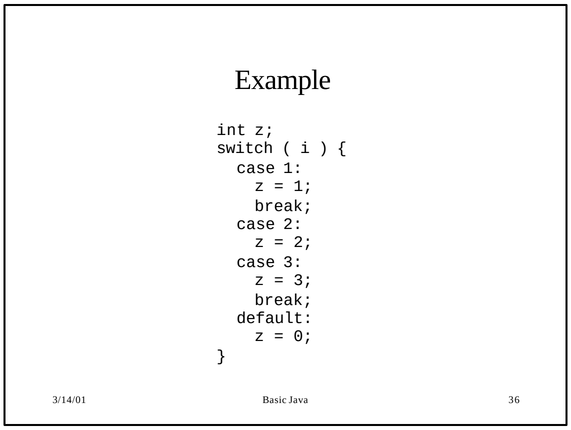


The switch statement

- Syntax:

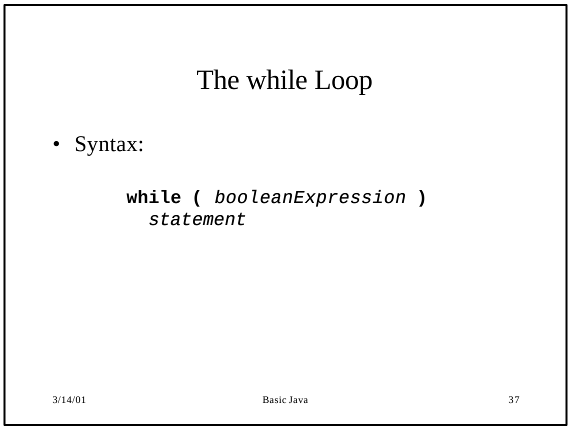
```
switch ( expression ) {
  case char/byte/short/int constant : statementSequence
  …
  default:  statementSequence
```

- As in C, break statements are needed to *jump* out of a switch statement.
- The default case is optional.



Example

```
int z;
switch ( i ) {
  case 1:
    z = 1;
    break;
  case 2:
    z = 2;
  case 3:
    z = 3;
    break;
  default:
    z = 0;
}
```

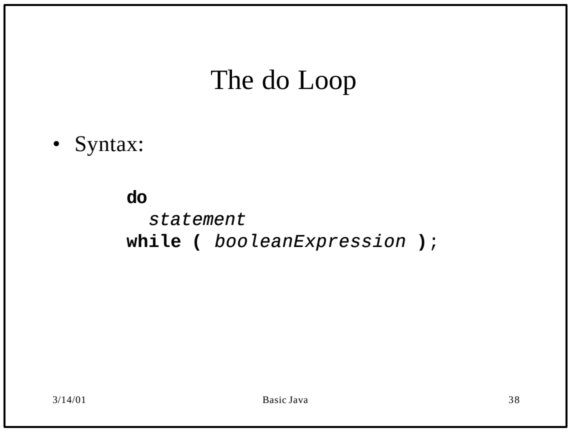


The while Loop

- Syntax:

```
while ( booleanExpression )
  statement
```

3/14/01 Basic Java 37

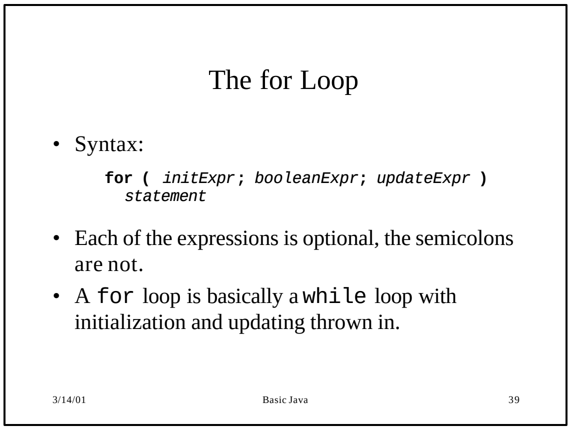


The do Loop

- Syntax:

```
do
  statement
while ( booleanExpression );
```

3/14/01 Basic Java 38



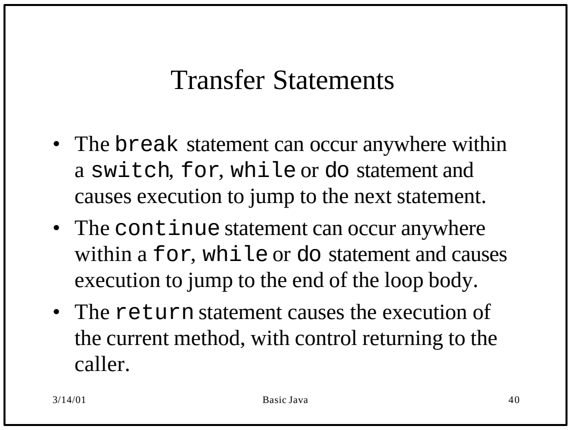
The for Loop

- Syntax:

```
for ( initExpr; booleanExpr; updateExpr )
  statement
```

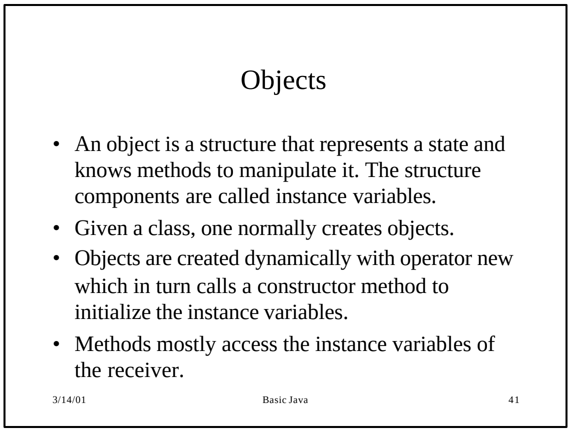
- Each of the expressions is optional, the semicolons are not.
- A `for` loop is basically a `while` loop with initialization and updating thrown in.

3/14/01 Basic Java 39



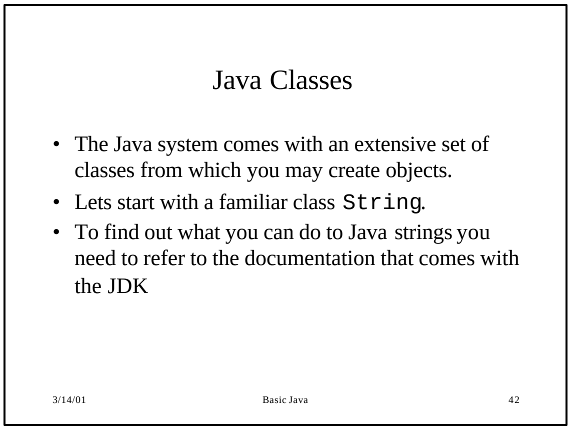
Transfer Statements

- The `break` statement can occur anywhere within a `switch`, `for`, `while` or `do` statement and causes execution to jump to the next statement.
- The `continue` statement can occur anywhere within a `for`, `while` or `do` statement and causes execution to jump to the end of the loop body.
- The `return` statement causes the execution of the current method, with control returning to the caller.



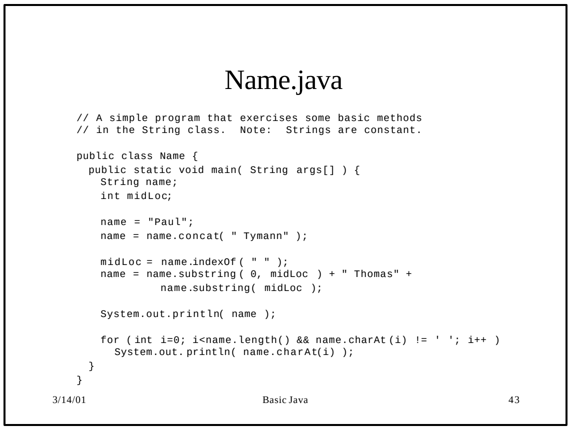
Objects

- An object is a structure that represents a state and knows methods to manipulate it. The structure components are called instance variables.
- Given a class, one normally creates objects.
- Objects are created dynamically with operator new which in turn calls a constructor method to initialize the instance variables.
- Methods mostly access the instance variables of the receiver.



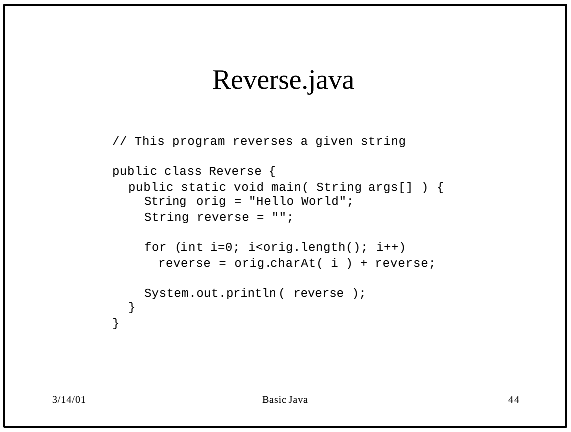
Java Classes

- The Java system comes with an extensive set of classes from which you may create objects.
- Lets start with a familiar class `String`.
- To find out what you can do to Java strings you need to refer to the documentation that comes with the JDK



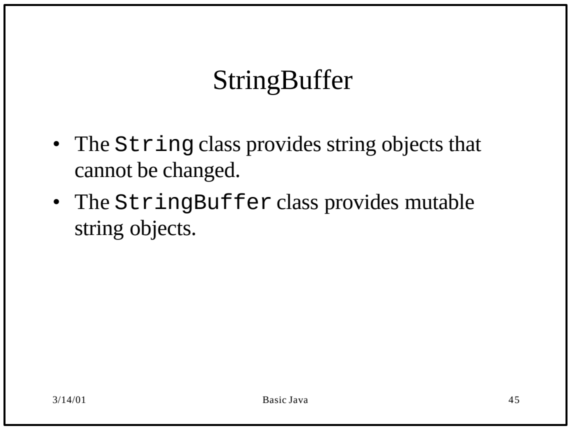
Name.java

```
// A simple program that exercises some basic methods
// in the String class.  Note:  Strings are constant.

public class Name {
  public static void main( String args[] ) {
    String name;
    int midLoc;

    name = "Paul";
    name = name.concat( " Tymann" );

    midLoc = name.indexOf( " " );
    name = name.substring( 0, midLoc ) + " Thomas" +
             name.substring( midLoc );

    System.out.println( name );

    for ( int i=0; i<name.length() && name.charAt(i) != ' '; i++ )
      System.out.println( name.charAt(i) );
  }
}
```



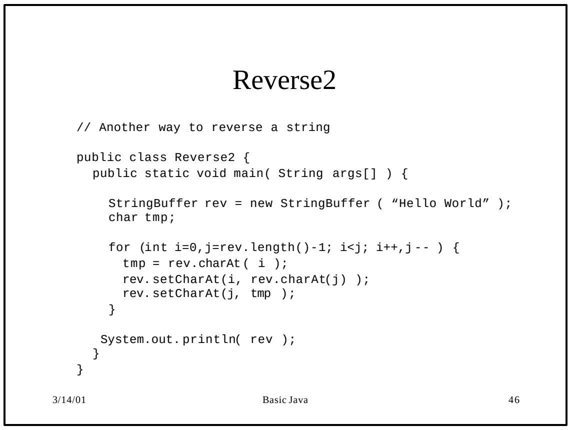
Reverse.java

```
// This program reverses a given string

public class Reverse {
  public static void main( String args[] ) {
    String orig = "Hello World";
    String reverse = "";

    for (int i=0; i<orig.length(); i++)
      reverse = orig.charAt( i ) + reverse;

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



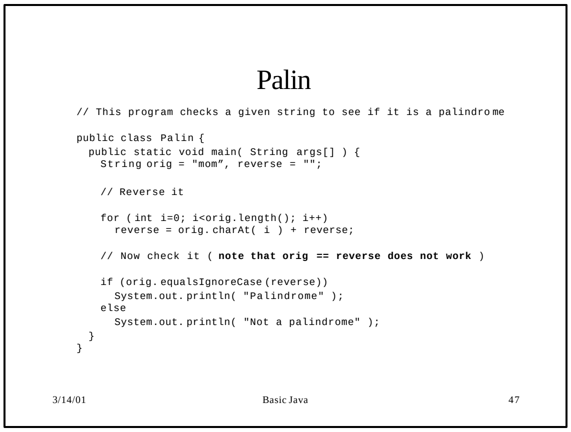
StringBuffer

- The `String` class provides string objects that cannot be changed.
- The `StringBuffer` class provides mutable string objects.



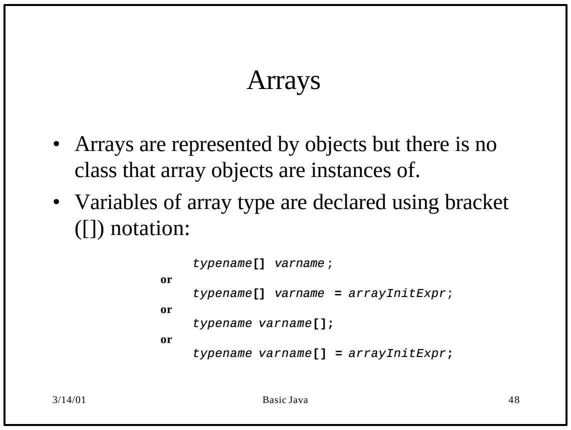
Reverse2

```
// Another way to reverse a string

public class Reverse2 {
  public static void main( String args[] ) {

    StringBuffer rev = new StringBuffer ( “Hello World” );
    char tmp;

    for (int i=0,j=rev.length()-1; i<j; i++,j-- ) {
      tmp = rev.charAt( i );
      rev.setCharAt(i, rev.charAt(j) );
      rev.setCharAt(j, tmp );
    }

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



Palin

```
// This program checks a given string to see if it is a palindrome

public class Palin {
  public static void main( String args[] ) {
    String orig = "mom”, reverse = "";

    // Reverse it

    for ( int i=0; i<orig.length(); i++)
      reverse = orig.charAt( i ) + reverse;

    // Now check it ( note that orig == reverse does not work )

    if (orig.equalsIgnoreCase(reverse))
      System.out.println( "Palindrome" );
    else
      System.out.println( "Not a palindrome" );
  }
}
```



Arrays

- Arrays are represented by objects but there is no class that array objects are instances of.
- Variables of array type are declared using bracket ([]) notation:

*typename***[]** *varname*;

or

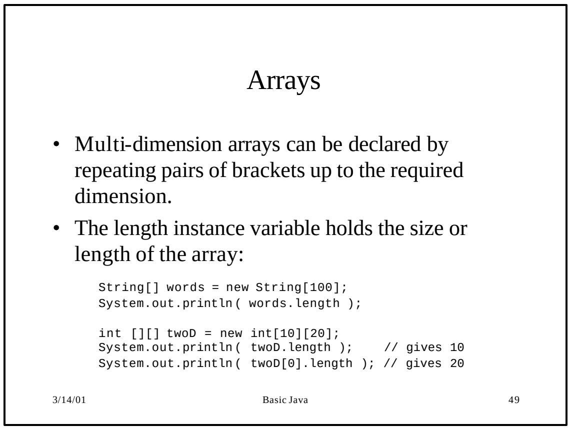
*typename***[]** *varname* **=** *arrayInitExpr*;

or

typename *varname***[];**

or

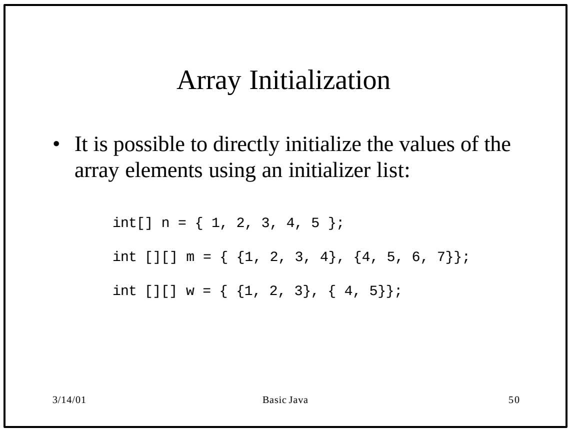
typename *varname***[] =** *arrayInitExpr*;



Arrays

- Multi-dimension arrays can be declared by repeating pairs of brackets up to the required dimension.
- The length instance variable holds the size or length of the array:

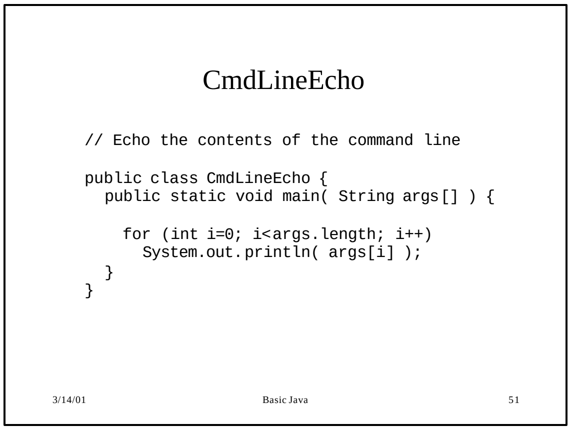
```
String[] words = new String[100];
System.out.println( words.length );

int [][] twoD = new int[10][20];
System.out.println( twoD.length );    // gives 10
System.out.println( twoD[0].length ); // gives 20
```



Array Initialization

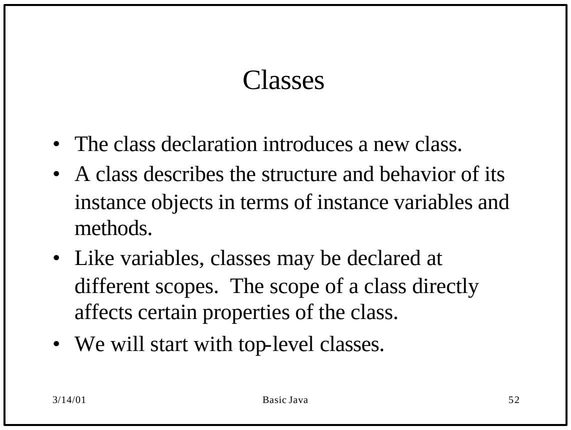
- It is possible to directly initialize the values of the array elements using an initializer list:

```
int[] n = { 1, 2, 3, 4, 5 };

int [][] m = { {1, 2, 3, 4}, {4, 5, 6, 7}};

int [][] w = { {1, 2, 3}, { 4, 5}};
```



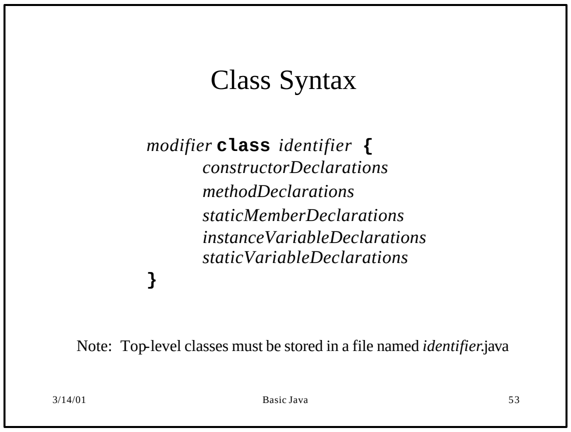
CmdLineEcho

```
// Echo the contents of the command line

public class CmdLineEcho {
  public static void main( String args[] ) {

    for (int i=0; i<args.length; i++)
      System.out.println( args[i] );
  }
}
```



Classes

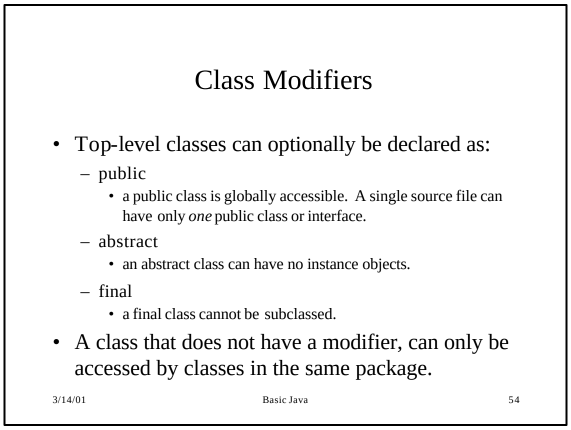
- The class declaration introduces a new class.
- A class describes the structure and behavior of its instance objects in terms of instance variables and methods.
- Like variables, classes may be declared at different scopes. The scope of a class directly affects certain properties of the class.
- We will start with top-level classes.



Class Syntax

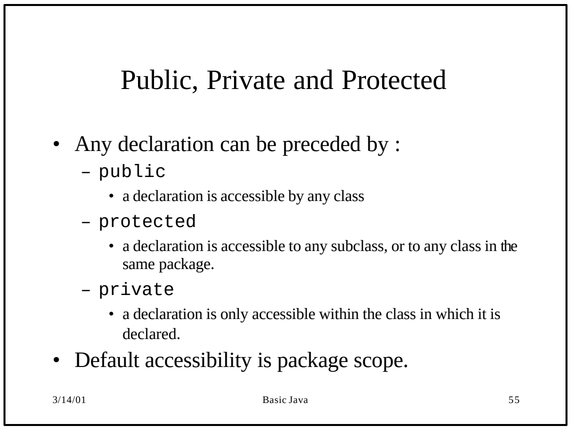
```
modifier class identifier {
        constructorDeclarations
        methodDeclarations
        staticMemberDeclarations
        instanceVariableDeclarations
        staticVariableDeclarations
}
```

Note: Top-level classes must be stored in a file named *identifier*.java



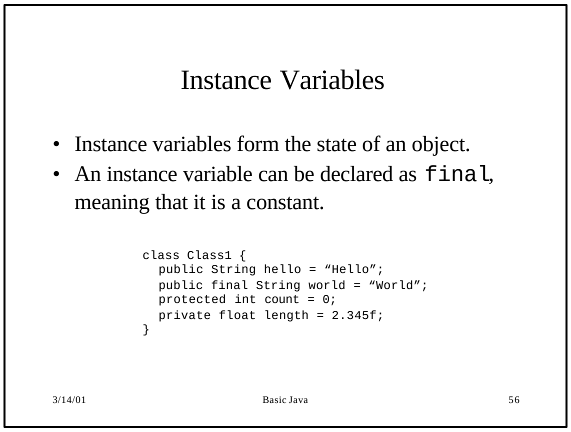
Class Modifiers

- Top-level classes can optionally be declared as:
  - public
    - a public class is globally accessible. A single source file can have only *one* public class or interface.
  - abstract
    - an abstract class can have no instance objects.
  - final
    - a final class cannot be subclassed.
- A class that does not have a modifier, can only be accessed by classes in the same package.



Public, Private and Protected

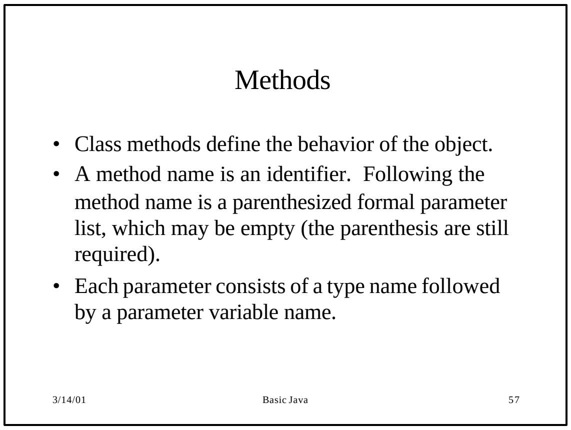
- Any declaration can be preceded by :
  - `public`
    - a declaration is accessible by any class
  - `protected`
    - a declaration is accessible to any subclass, or to any class in the same package.
  - `private`
    - a declaration is only accessible within the class in which it is declared.
- Default accessibility is package scope.



Instance Variables

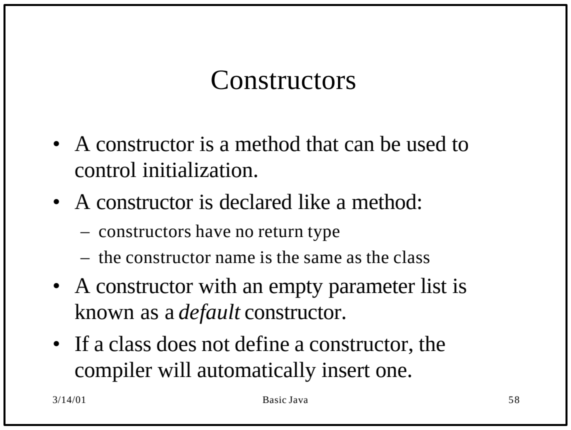
- Instance variables form the state of an object.
- An instance variable can be declared as `final`, meaning that it is a constant.

```
class Class1 {
  public String hello = “Hello”;
  public final String world = “World”;
  protected int count = 0;
  private float length = 2.345f;
}
```



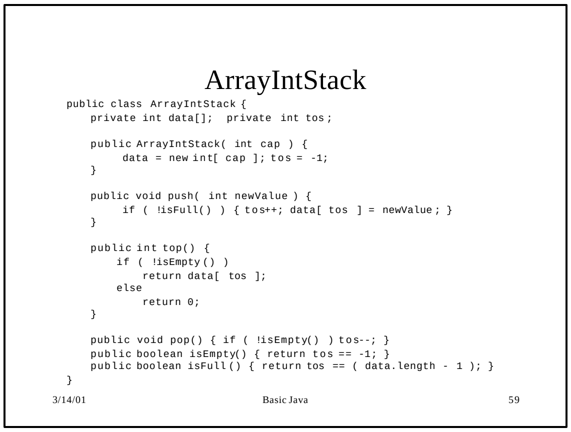
Methods

- Class methods define the behavior of the object.
- A method name is an identifier. Following the method name is a parenthesized formal parameter list, which may be empty (the parenthesis are still required).
- Each parameter consists of a type name followed by a parameter variable name.



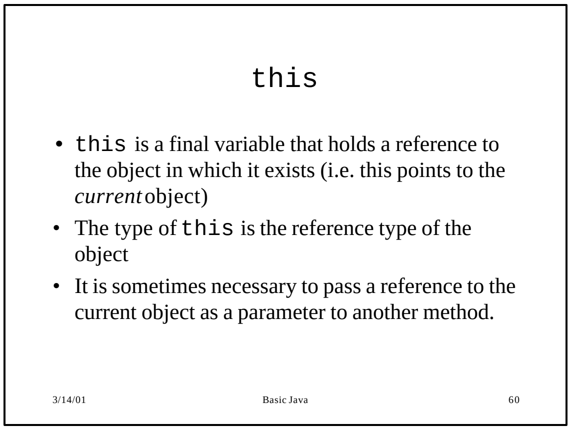
Constructors

- A constructor is a method that can be used to control initialization.
- A constructor is declared like a method:
  - constructors have no return type
  - the constructor name is the same as the class
- A constructor with an empty parameter list is known as a *default* constructor.
- If a class does not define a constructor, the compiler will automatically insert one.



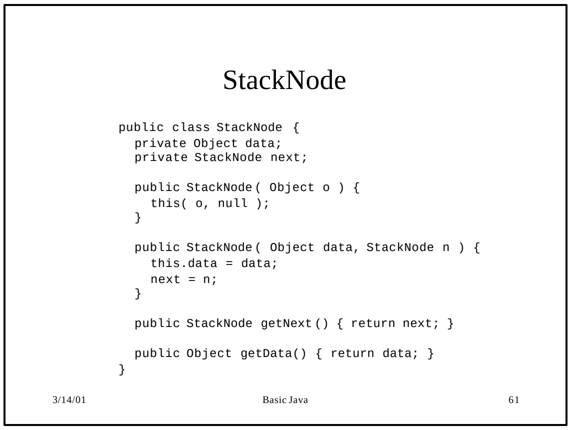
ArrayIntStack

```
public class ArrayIntStack {
    private int data[];  private int tos;

    public ArrayIntStack( int cap ) {
         data = new int[ cap ]; tos = -1;
    }

    public void push( int newValue ) {
         if ( !isFull() ) { tos++; data[ tos ] = newValue; }
    }

    public int top() {
        if ( !isEmpty() )
            return data[ tos ];
        else
            return 0;
    }

    public void pop() { if ( !isEmpty() ) tos--; }
    public boolean isEmpty() { return tos == -1; }
    public boolean isFull() { return tos == ( data.length - 1 ); }
}
```



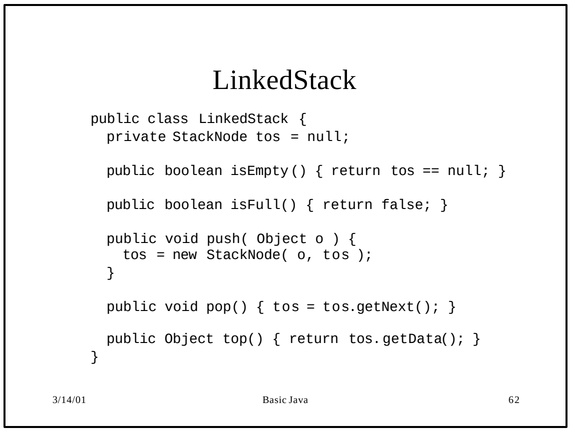
`this`

- `this` is a final variable that holds a reference to the object in which it exists (i.e. this points to the *current* object)
- The type of `this` is the reference type of the object
- It is sometimes necessary to pass a reference to the current object as a parameter to another method.



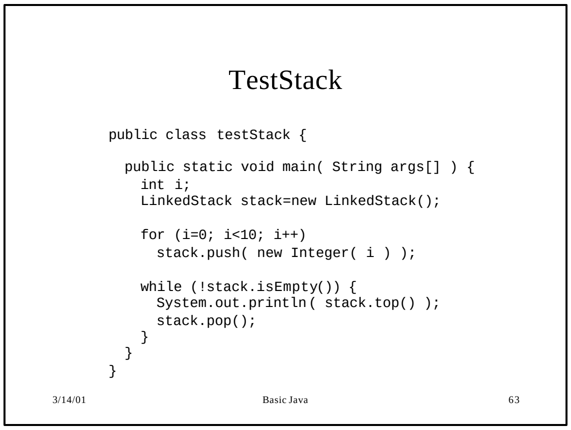
StackNode

```
public class StackNode {
  private Object data;
  private StackNode next;

  public StackNode( Object o ) {
    this( o, null );
  }

  public StackNode( Object data, StackNode n ) {
    this.data = data;
    next = n;
  }

  public StackNode getNext() { return next; }

  public Object getData() { return data; }
}
```



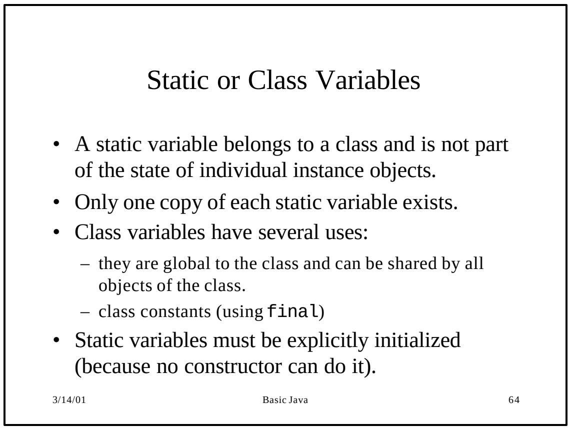
LinkedStack

```
public class LinkedStack {
  private StackNode tos = null;

  public boolean isEmpty() { return tos == null; }

  public boolean isFull() { return false; }

  public void push( Object o ) {
    tos = new StackNode( o, tos );
  }

  public void pop() { tos = tos.getNext(); }

  public Object top() { return tos.getData(); }
}
```



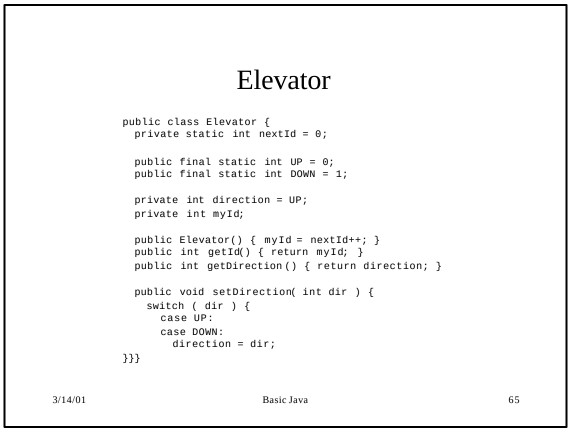
TestStack

```
public class testStack {

  public static void main( String args[] ) {
    int i;
    LinkedStack stack=new LinkedStack();

    for (i=0; i<10; i++)
      stack.push( new Integer( i ) );

    while (!stack.isEmpty()) {
      System.out.println( stack.top() );
      stack.pop();
    }
  }
}
```



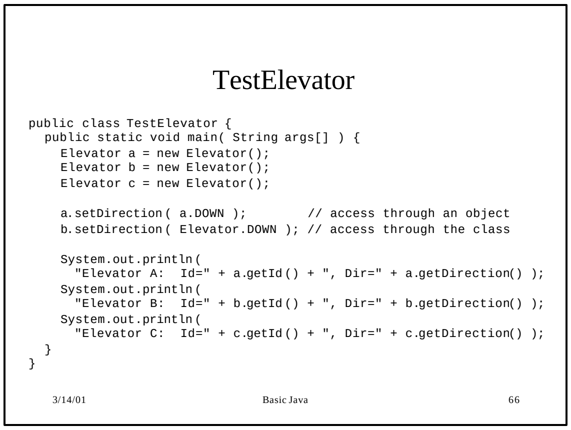
Static or Class Variables

- A static variable belongs to a class and is not part of the state of individual instance objects.
- Only one copy of each static variable exists.
- Class variables have several uses:
  - they are global to the class and can be shared by all objects of the class.
  - class constants (using `final`)
- Static variables must be explicitly initialized (because no constructor can do it).



Elevator

```
public class Elevator {
  private static int nextId = 0;

  public final static int UP = 0;
  public final static int DOWN = 1;

  private int direction = UP;
  private int myId;

  public Elevator() { myId = nextId++; }
  public int getId() { return myId; }
  public int getDirection() { return direction; }

  public void setDirection( int dir ) {
    switch ( dir ) {
      case UP:
      case DOWN:
        direction = dir;
}}}
```



TestElevator

```
public class TestElevator {
  public static void main( String args[] ) {
    Elevator a = new Elevator();
    Elevator b = new Elevator();
    Elevator c = new Elevator();

    a.setDirection( a.DOWN );        // access through an object
    b.setDirection( Elevator.DOWN ); // access through the class

    System.out.println(
      "Elevator A:  Id=" + a.getId() + ", Dir=" + a.getDirection() );
    System.out.println(
      "Elevator B:  Id=" + b.getId() + ", Dir=" + b.getDirection() );
    System.out.println(
      "Elevator C:  Id=" + c.getId() + ", Dir=" + c.getDirection() );
  }
}
```

Static Methods

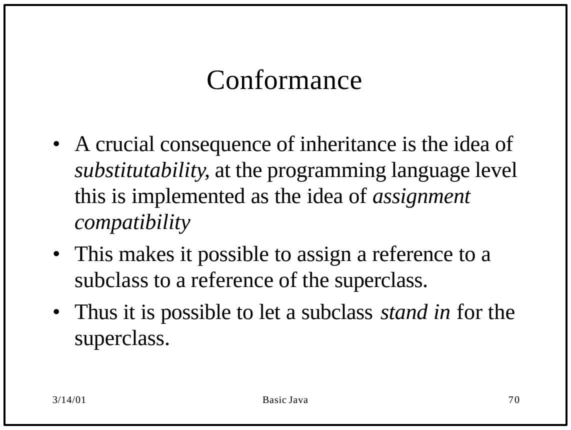
- Static methods generally follow the same rules as methods:
  - a static method belongs to a class not its instance objects.
  - a static method can be called both directly and for an object of the same class
  - a static method cannot access any instance variables or methods (since it does not belong to an instance object)
  - `this` cannot be used

Static Methods

- There is one special use of static methods in the form of `static main.`
- When a class defines a public static method `main`, it provides a starting point for execution of a program using that class.
- Any class can have a static `main` method.
- Static methods are generally used to provide utility or helper methods. For examples see `java.lang.Math`.

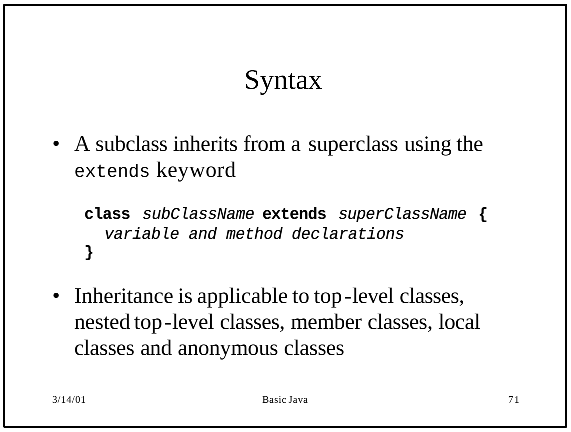
Inheritance

- Inheritance provides a mechanism for extending an existing class to create a new class.
- The new class has all the features of the old class and adds its own features.
- The class that inherits is known as the *subclass*, while the class that is inherited from is known as the *superclass*.



Conformance

- A crucial consequence of inheritance is the idea of *substitutability*, at the programming language level this is implemented as the idea of *assignment compatibility*
- This makes it possible to assign a reference to a subclass to a reference of the superclass.
- Thus it is possible to let a subclass *stand in* for the superclass.

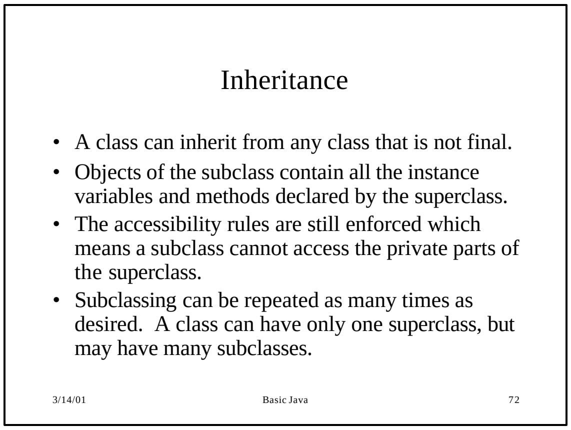


Syntax

- A subclass inherits from a superclass using the `extends` keyword

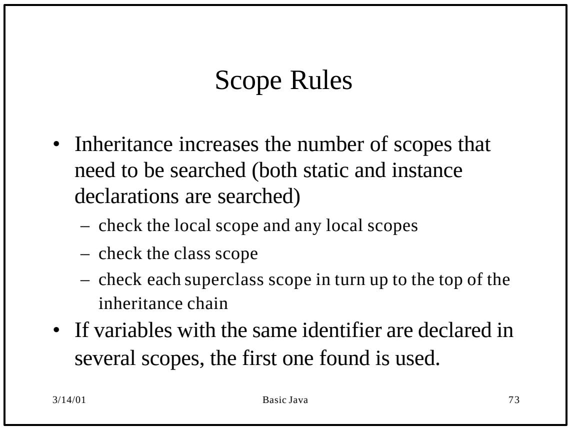
```
class subClassName extends superClassName {
  variable and method declarations
}
```

- Inheritance is applicable to top-level classes, nested top-level classes, member classes, local classes and anonymous classes



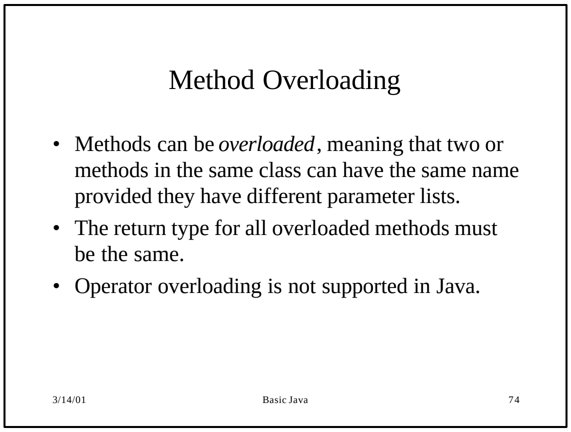
Inheritance

- A class can inherit from any class that is not final.
- Objects of the subclass contain all the instance variables and methods declared by the superclass.
- The accessibility rules are still enforced which means a subclass cannot access the private parts of the superclass.
- Subclassing can be repeated as many times as desired. A class can have only one superclass, but may have many subclasses.



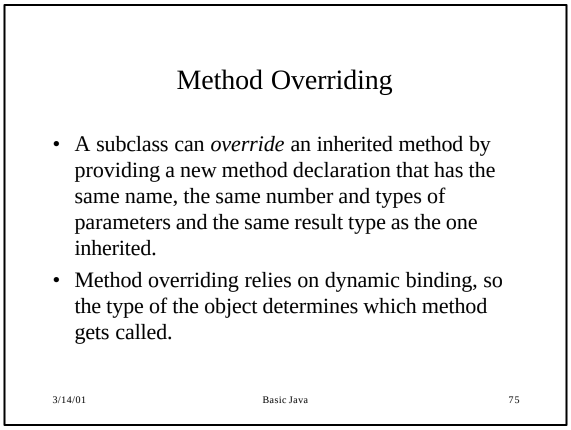
Scope Rules

- Inheritance increases the number of scopes that need to be searched (both static and instance declarations are searched)
  - check the local scope and any local scopes
  - check the class scope
  - check each superclass scope in turn up to the top of the inheritance chain
- If variables with the same identifier are declared in several scopes, the first one found is used.



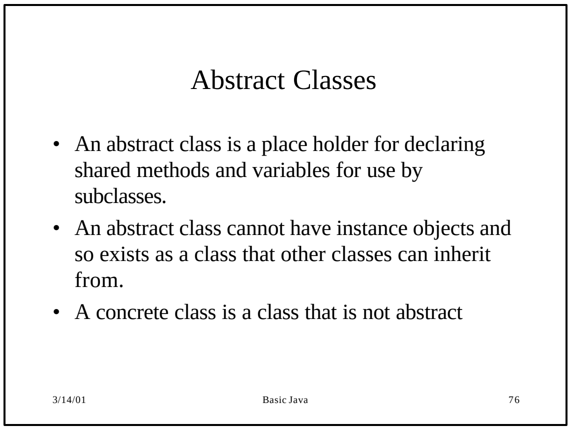
Method Overloading

- Methods can be *overloaded*, meaning that two or methods in the same class can have the same name provided they have different parameter lists.
- The return type for all overloaded methods must be the same.
- Operator overloading is not supported in Java.



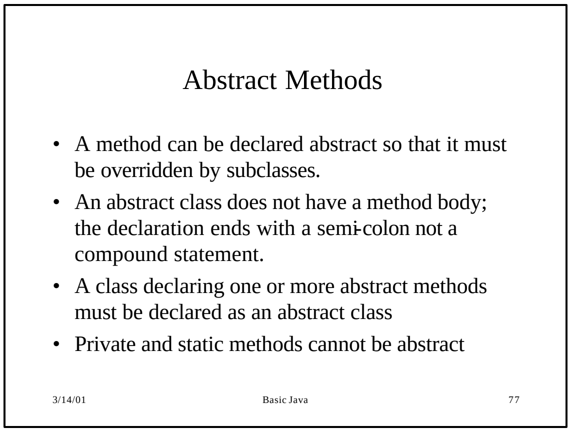
Method Overriding

- A subclass can *override* an inherited method by providing a new method declaration that has the same name, the same number and types of parameters and the same result type as the one inherited.
- Method overriding relies on dynamic binding, so the type of the object determines which method gets called.



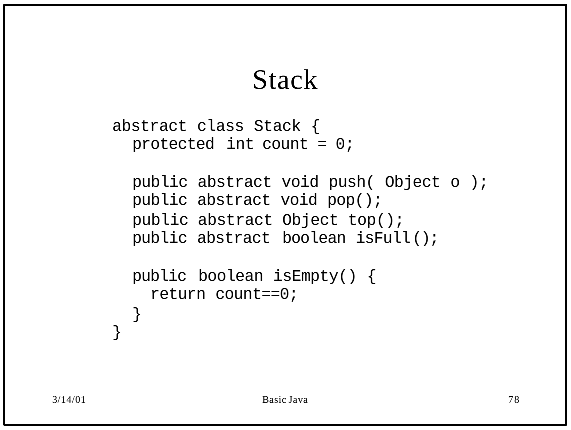
Abstract Classes

- An abstract class is a place holder for declaring shared methods and variables for use by subclasses.
- An abstract class cannot have instance objects and so exists as a class that other classes can inherit from.
- A concrete class is a class that is not abstract



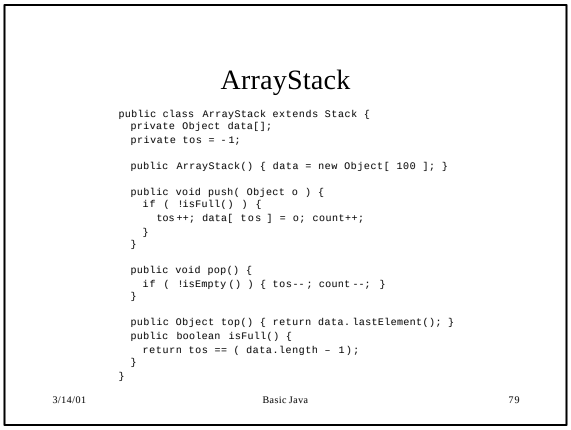
Abstract Methods

- A method can be declared abstract so that it must be overridden by subclasses.
- An abstract class does not have a method body; the declaration ends with a semi-colon not a compound statement.
- A class declaring one or more abstract methods must be declared as an abstract class
- Private and static methods cannot be abstract



Stack

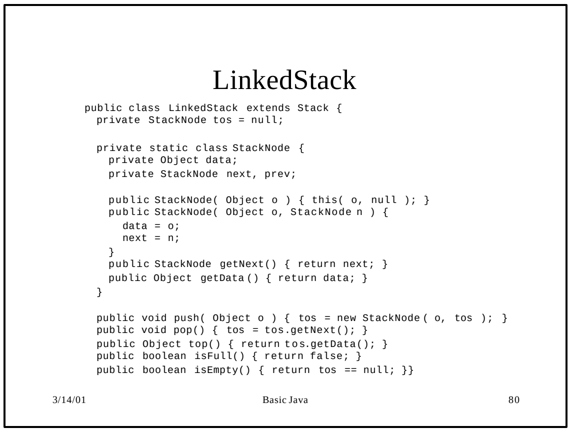
```
abstract class Stack {
  protected int count = 0;

  public abstract void push( Object o );
  public abstract void pop();
  public abstract Object top();
  public abstract boolean isFull();

  public boolean isEmpty() {
    return count==0;
  }
}
```



ArrayStack

```
public class ArrayStack extends Stack {
  private Object data[];
  private tos = -1;

  public ArrayStack() { data = new Object[ 100 ]; }

  public void push( Object o ) {
    if ( !isFull() ) {
      tos++; data[ tos ] = o; count++;
    }
  }

  public void pop() {
    if ( !isEmpty() ) { tos--; count--; }
  }

  public Object top() { return data.lastElement(); }
  public boolean isFull() {
    return tos == ( data.length – 1);
  }
}
```

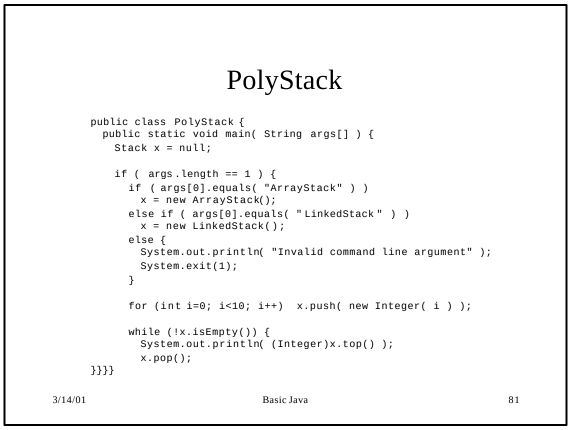
3/14/01 Basic Java 79



LinkedStack

```
public class LinkedStack extends Stack {
  private StackNode tos = null;

  private static class StackNode {
    private Object data;
    private StackNode next, prev;

    public StackNode( Object o ) { this( o, null ); }
    public StackNode( Object o, StackNode n ) {
      data = o;
      next = n;
    }
    public StackNode getNext() { return next; }
    public Object getData() { return data; }
  }

  public void push( Object o ) { tos = new StackNode( o, tos ); }
  public void pop() { tos = tos.getNext(); }
  public Object top() { return tos.getData(); }
  public boolean isFull() { return false; }
  public boolean isEmpty() { return tos == null; }}
```

3/14/01 Basic Java 80



PolyStack

```
public class PolyStack {
  public static void main( String args[] ) {
    Stack x = null;

    if ( args.length == 1 ) {
      if ( args[0].equals( "ArrayStack" ) )
        x = new ArrayStack();
      else if ( args[0].equals( "LinkedStack" ) )
        x = new LinkedStack();
      else {
        System.out.println( "Invalid command line argument" );
        System.exit(1);
      }

      for (int i=0; i<10; i++)  x.push( new Integer( i ) );

      while (!x.isEmpty()) {
        System.out.println( (Integer)x.top() );
        x.pop();
}}}}
```

3/14/01 Basic Java 81

Final Methods

- A final instance method cannot be overridden (but can still be overloaded).
- A final static method cannot be re-declared in a sublcass.
- Final methods prevent a method that has the same name and parameter types from being declared in a subclass.
- This takes into account both static and instance variables.

3/14/01 Basic Java 82

Constructors and Inheritance

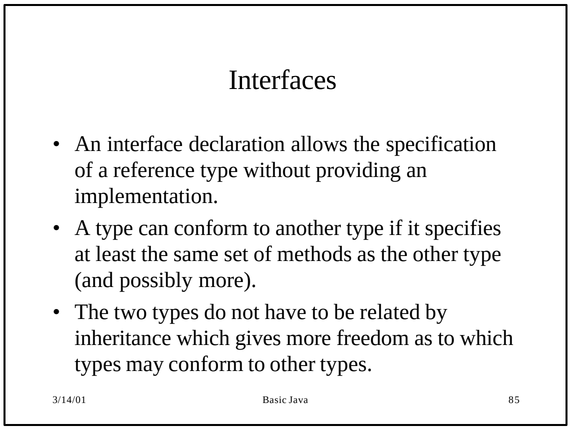
- The guarantee of proper initialization must be maintained in the presence of inheritance.
- Java forces the constructors for each superclass to be called and provides syntax for explicitly controlling which constructors are called.
- The keyword `super` can be used to explicitly call a superclass constructor
  - **`super (`** ***`argumentList`*** **`) ;`**
- `super` must be the first statement in a constructor

3/14/01 Basic Java 83

Methods Inherited from Class Object

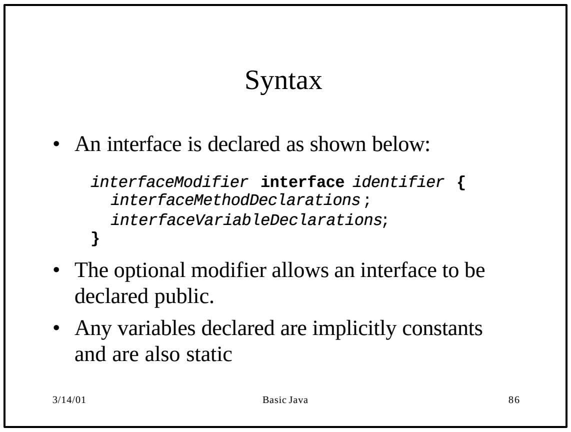
- Class Object declares the following methods that can be overwritten:
  - public boolean equals(Object obj);
  - public String toString();
  - public final native int hashCode() ;
  - protective native Object clone();
  - protected void finalize();
  - public final Class getClass()

3/14/01 Basic Java 84



Interfaces

- An interface declaration allows the specification of a reference type without providing an implementation.
- A type can conform to another type if it specifies at least the same set of methods as the other type (and possibly more).
- The two types do not have to be related by inheritance which gives more freedom as to which types may conform to other types.

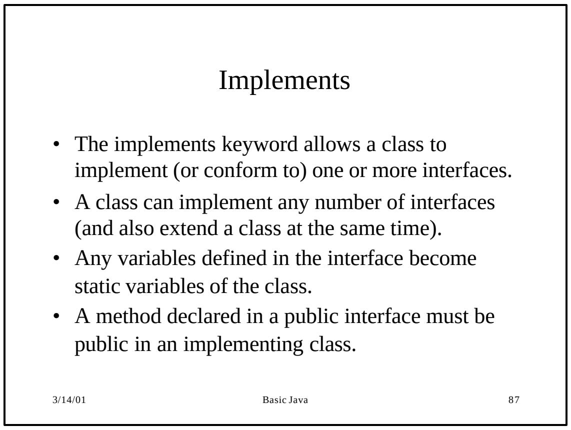


Syntax

- An interface is declared as shown below:

```
interfaceModifier interface identifier {
  interfaceMethodDeclarations;
  interfaceVariableDeclarations;
}
```

- The optional modifier allows an interface to be declared public.
- Any variables declared are implicitly constants and are also static



Implements

- The implements keyword allows a class to implement (or conform to) one or more interfaces.
- A class can implement any number of interfaces (and also extend a class at the same time).
- Any variables defined in the interface become static variables of the class.
- A method declared in a public interface must be public in an implementing class.