

Crash Course in Java



Based on notes from Dennis Frey, Susan Mitchell, John Park, D. Hollinger and J.J. Johns, and material from *Java in a Nutshell* and *Java Network Programming and Distributed Computing*

Java History

- Created by **Sun Microsystems** team led by **James Gosling** (1991)
- Originally designed for programming home appliances
 - Difficult task because appliances are controlled by a wide variety of computer processors
 - Writing a compiler (translation program) for each type of appliance processor would have been very costly.
 - Solution: **two-step translation process**
 - compile, then
 - interpret

First Program

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

Python vs. Java – A Little Sample

Python:

```
print "Hello, world"
quotient = 3 / 4
if quotient == 0:
    print "3/4 == 0",
    print "in Python"
else:
    print "3/4 != 0"
```

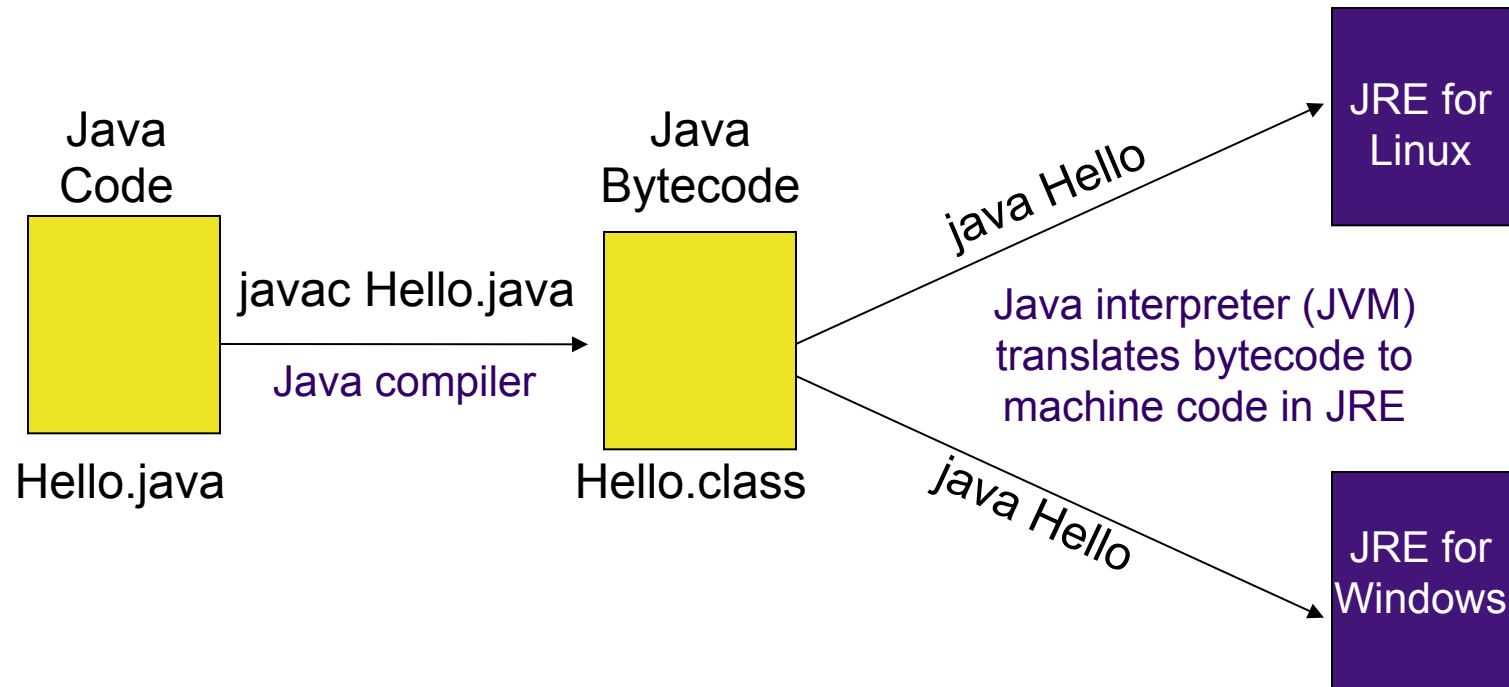
Java:

```
public class Hello {
    public static void main(String[] args) {
        int quotient;

        System.out.println("Hello, world");
        quotient = 3 / 4;
        if (quotient == 0) {
            System.out.print("3/4 == 0");
            System.out.println(" in Java");
        }
        else {
            System.out.println("3/4 != 0");
        }
    }
}

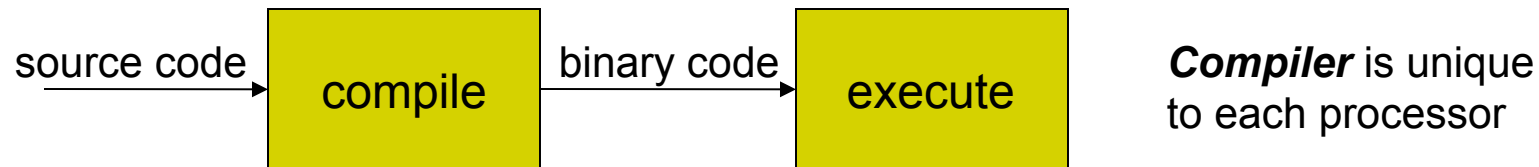
// Things to note:
// Everything has to be in some class
// We need a "main()"
// Statements end with ';'
// Variables must be declared
// "if/else" syntax different
// Statement blocks demarcated by "{...}"
// Comments are different ☺
// ...but there is much that is similar
```

Compiling and Running Java

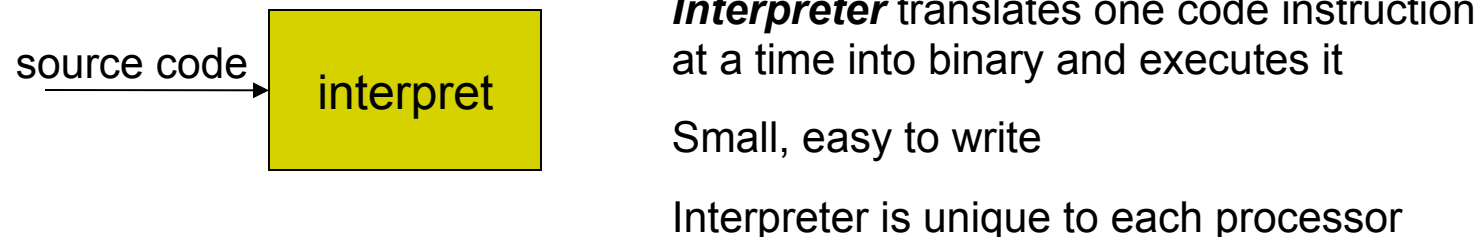


Compilers, Interpreters, and the JVM

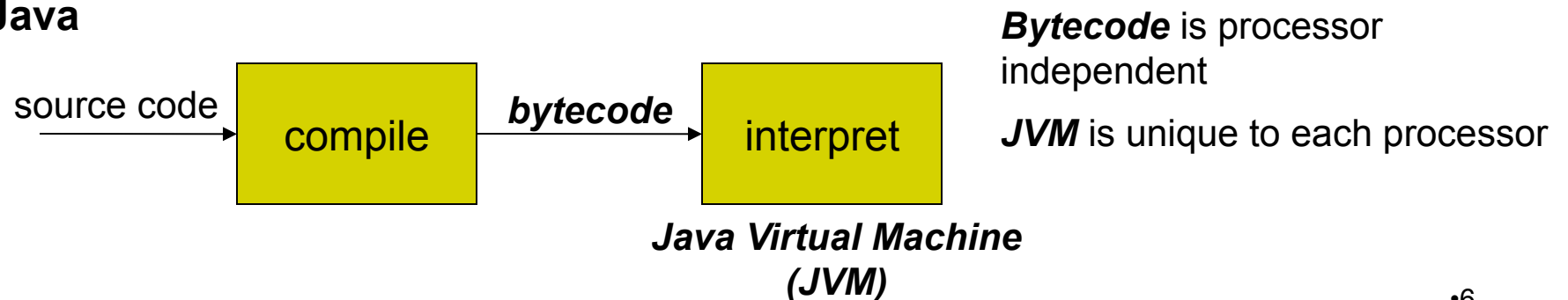
Compiled Languages (e.g. C, C++)



Interpreted Languages (e.g. JavaScript, Perl, Ruby)



Java



Java Terminology

Java acronyms are plentiful and confusing. Here are the basics.

- **JVM** – Java Virtual Machine
 - Translates Java bytecode to machine code
- **API** – Application Programming Interface
 - Java code libraries
- **JRE** – Java Runtime Environment
 - The JVM and the Java API together
- **JDK** (formerly SDK) – Java Development Kit
 - JRE + tools (compiler, debugger) for developing Java applications and applets
- **J2SE** – Java 2 Platform, Standard Edition
 - The JRE and JDK products taken as a “family”
- To learn more about JDK, JRE, etc, visit:

<http://java.sun.com/javase/technologies/index.jsp>

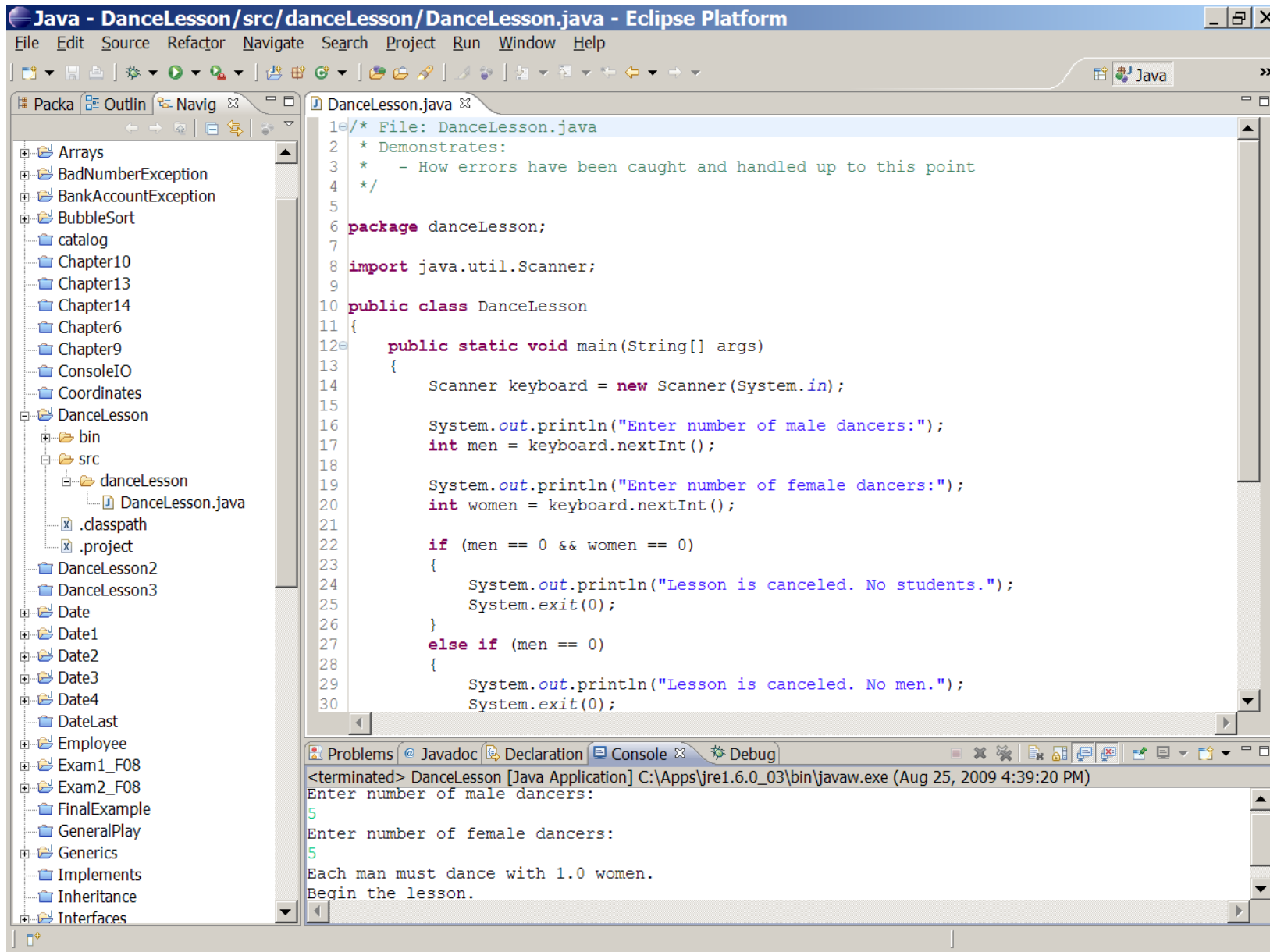
Java Versions

- Current version of Java: Java 7, also known as Java 1.7 or Java 1.7.0
- Previous version: Java 6, also known as Java 1.6, Java 1.6.0 or “Java 2 SE Version 6”
- To learn more about Java version naming, visit:
<http://java.sun.com/javase/namechange.html>

The Eclipse IDE

- An ***integrated development environment (IDE)*** for writing Java programs. Contains (minimally):
 - editor
 - debugger
 - Java compiler
 - Java JVM
- Free download for your PC (link on course website)
- Available in the computing labs
- We'll show you more later

Eclipse IDE Screenshot



Java Basics

Simple “Procedural” Java

```
public class MyClass {
    static boolean sawNonZero = false;

    public static void main(String[] args) {
        System.out.print("Hello, world");

        int quotient = 3 / 4;
        if (testNonZero(quotient)) {
            System.out.print("\nQuotient is non-zero\n");
        }
    }

    static boolean testNonZero(int value) {
        if (value != 0) {
            sawNonZero = true;
            return true;
        } else
            return false;
    }
}
```

Java Program Basics

- All code has to be inside some class definition
 - For now, we can think of this like in terms of file/module, or namespace
- All programs begin execution at **main()**
 - This is much like in C, but...
 - You can have a different main() in every class: pick at runtime
- **System.out.print()**
 - Outputs text to the screen
 - `System.out.print("Hello");`
 - There is also **System.out.println()**, which terminates w/newline
- Can program procedurally:
 - Just put the word “static” in front of all functions and global variables.

Variable Declarations

- Format: *type variable-name;*
- Examples:
 - `int total;`
 - `float salary;`
- Variables may be declared anywhere in the code, but may not be used until declared.
 - Note the declaration of `int quotient;` in the sample program.
 - This feature allows you to declare variables close to where they are used, making code more readable.
 - However, “can” doesn’t imply “should”—in general, declarations are often best at top of a block

Variable Declarations (con' t)

- When we declare a variable, we tell Java:
 - When and where to set aside memory space for the variable
 - How much memory to set aside
 - How to interpret the contents of that memory: the specified data **type**
 - What name we will be referring to that location by: its **identifier**

Naming Conventions

- Variables, methods, and objects
 - Start with a lowercase letter
 - Indicate "word" boundaries with an uppercase letter
 - Restrict the remaining characters to digits and lowercase letters
 - Can use underscores

`topSpeed` `bankRate1` `timeOfArrival`

- Classes
 - Start with an uppercase letter
 - Otherwise, adhere to the rules above

`FirstProgram` `MyClass` `String`

Primitive Types

Display 1.2 **Primitive Types**

TYPE NAME	KIND OF VALUE	MEMORY USED	SIZE RANGE
boolean	true or false	1 byte	not applicable
char	single character (Unicode)	2 bytes	all Unicode characters
byte	integer	1 byte	−128 to 127
short	integer	2 bytes	−32768 to 32767
int	integer	4 bytes	−2147483648 to 2147483647
long	integer	8 bytes	−9223372036854775808 to 9223372036854775807
float	floating-point number	4 bytes	$-3.40282347 \times 10^{+38}$ to $-1.40239846 \times 10^{-45}$
double	floating-point number	8 bytes	$\pm 1.76769313486231570 \times 10^{+308}$ to $\pm 4.94065645841246544 \times 10^{-324}$

Fixed Size for Primitive Types

- Java byte-code runs on the Java Virtual Machine (JVM).
 - Therefore, the size (number of bytes) for each primitive type is fixed.
 - The size is not dependent on the actual machine/device on which the code executes.
 - The machine-specific JVM is responsible for mapping Java primitive types to native types on the particular architecture

Operators

- Assignment: `=, +=, -=, *=, /=, %=...`
- Numeric: `+, -, *, /, %, ++, --, ...`
- Relational: `==, !=, <, >, <=, >=, ...`
- Boolean: `&&, ||, !`
- Bitwise: `&, |, ^, ~, <<, >>, ...`

Arithmetic Operators

Rules of Operator Precedence

<u>Operator(s)</u>	<u>Precedence & Associativity</u>
()	Evaluated first. If nested , innermost first. If on same level, left to right.
* / %	Evaluated second. If there are several, evaluated left to right.
+ -	Evaluated third. If there are several, evaluated left to right.
=	Evaluated last, right to left.

Practice With Evaluating Expressions

Given integer variables a, b, c, d, and e,
where $a = 1$, $b = 2$, $c = 3$, $d = 4$,
evaluate the following expressions:

$$a + b - c + d$$

$$a * b / c$$

$$1 + a * b \% c$$

$$a + d \% b - c$$

$$e = b = d + c / b - a$$

A Hand Trace Example

```
int answer, value = 4 ;
```

<u>Code</u>	<u>Value</u>	<u>Answer</u>
	4	garbage

```
value = value + 1 ;
```

```
value++ ;
```

```
++value ;
```

```
answer = 2 * value++ ;
```

```
answer = ++value / 2 ;
```

```
value-- ;
```

```
--value ;
```

```
answer = --value * 2 ;
```

```
answer = value-- / 3 ;
```

More Practice

Given

```
int a = 1, b = 2, c = 3, d = 4 ;
```

What is the value of this expression?

```
++b / c + a * d++
```

What are the new values of a, b, c, and d?

Assignment Operators

= += -= *= /= %=

Statement

Equivalent Statement

a = a + 2 ;

a += 2 ;

a = a - 3 ;

a -= 3 ;

a = a * 2 ;

a *= 2 ;

a = a / 4 ;

a /= 4 ;

a = a % 2 ;

a %= 2 ;

b = b + (c + 2) ;

b += c + 2 ;

d = d * (e - 5) ;

d *= e - 5 ;

Type Casting

- A ***type cast*** takes a value of one type and produces a value of another type with an "equivalent" value.

```
int n, m;
```

```
double ans = n / (double)m;
```

OR

```
double ans = (double)n / m;
```

OR

```
double ans = (double)n / (double)m;
```

- The type and value of **n** and **m** do not change.

Java Comparison Operators

Display 3.3 Java Comparison Operators

MATH NOTATION	NAME	JAVA NOTATION	JAVA EXAMPLES
=	Equal to	==	<code>x + 7 == 2*y</code> <code>answer == 'y'</code>
≠	Not equal to	!=	<code>score != 0</code> <code>answer != 'y'</code>
>	Greater than	>	<code>time > limit</code>
≥	Greater than or equal to	>=	<code>age >= 21</code>
<	Less than	<	<code>pressure < max</code>
≤	Less than or equal to	<=	<code>time <= limit</code>

Boolean Expressions

- Operators: &&, ||, !
- Boolean expression evaluates to the values **true** or **false**
- Simple Boolean expressions:

time < limit

yourScore == myScore

- Two equal signs (**==**): equality testing
- Single equal sign (**=**): assignment

Control Structures

Java Flow Control

- Decisions
if, if-else, switch
- Loops
for, while, do-while
- Boolean expressions
 - Java flow control constructs evaluate Boolean expressions
 - The expression *must* be of boolean type:
 - Cannot do: “if (--c)...”; must do: “if (--c != 0)...”

if-else & while Statements

```
if ( condition1 ) {  
    statement(s)  
} else if ( condition2 ) {  
    statement(s)  
}  
    ...          /* more else if clauses may be here */  
} else {  
    statement(s)  /* the default case */  
}
```

```
while ( condition ) {  
    statement(s)  
}
```

Example

```
while ( children > 0 ) {  
    children = children - 1 ;  
    cookies = cookies * 2 ;  
}
```

Good Programming Practice

- Always place braces around the bodies of the if and else clauses of an if-else statement.
- Advantages:
 - Easier to read
 - Will not forget to add the braces if you go back and add a second statement to the clause
 - Less likely to make a semantic error
- Indent the bodies of the if and else clauses 3 to 4 spaces -- be consistent!

Example

```
...  
factorial = 1;  
while ( myNumber > 0 ) {  
    factorial *= myNumber;  
    --myNumber;  
}  
return factorial;
```

The 3 Parts of a Loop

...

int i = 1 ;  initialization of **loop control variable**

// count from 1 to 100

while (i < 101) {  test of **loop termination condition**

System.out.println(i) ;

i = i + 1 ;  modification of loop control variable

}

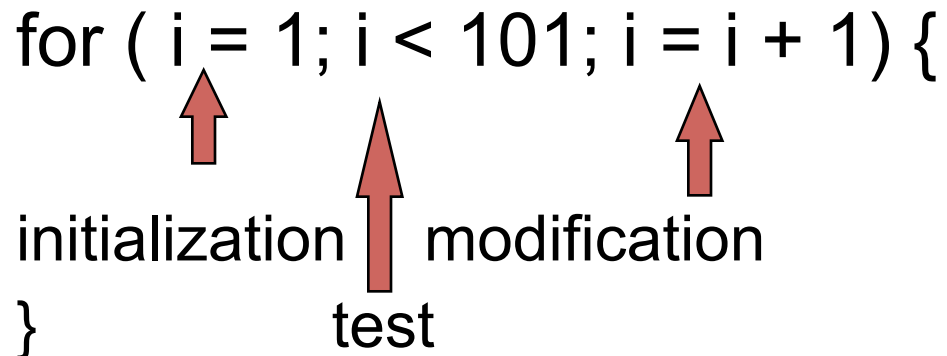
return 0 ;

}

The for Loop Repetition Structure

- The **for** loop handles details of the counter-controlled loop “automatically”.
- The initialization of the the loop control variable, the termination condition test, and control variable modification are handled in the **for** loop structure.

```
for ( i = 1; i < 101; i = i + 1 ) {  
    initialization  
    test  
    modification  
}
```



When Does a for Loop Initialize, Test and Modify?

- Just as with a while loop, a for loop
 - initializes the loop control variable before beginning the first loop iteration
 - performs the loop termination test before each iteration of the loop
 - modifies the loop control variable at the very end of each iteration of the loop
- The for loop is easier to write and read for counter-controlled loops.

for Loop Examples

- A *for* loop that counts from 0 to 9:

```
// modify part can be simply "i++"  
for ( i = 0; i < 10; i = i + 1 ) {  
    System.out.println( i ) ;  
}
```

- ...or we can count backwards by 2' s :

```
// modify part can be "i -= 2"  
for ( i = 10; i > 0; i = i - 2 ) {  
    System.out.println( i ) ;  
}
```

The do-while Repetition Structure

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

- The body of a **do-while** is ALWAYS executed at least once. Is this true of a **while** loop? What about a **for** loop?

The *break* & *continue* Statements

- The `break` & `continue` statements can be used in **while**, **do-while**, and **for** loops to cause the remaining statements in the body of the loop to be skipped; then:
 - `break` causes the looping itself to abort, while...
 - `continue` causes the next turn of the loop to start. In a **for** loop, the modification step will still be executed.

Example break in a for Loop

```
...
int i ;
for (i = 1; i < 10; i = i + 1) {
    if (i == 5) {
        break;
    }
    System.out.println(i);
}
System.out.println("\nBroke out of loop at i = " + i);
```

•OUTPUT:

• 1 2 3 4

•Broke out of loop at i = 5.

Example continue in a for Loop

```
...
int i;
for (i = 1; i < 10; i = i + 1) {
    if (i == 5) {
        continue;
    }
    System.out.println(i);
}
System.out.println("Done");
```

OUTPUT:

1 2 3 4 6 7 8 9

Done.

Problem: continue in while Loop

```
// This seems equivalent to for loop  
// in previous slide—but is it??
```

OUTPUT:

```
...
```

???

```
int i = 1;  
while (i < 10) {  
    if (i == 5) {  
        continue;  
    }  
    System.out.println(i);  
    i = i + 1;  
}  
System.out.println("Done");
```

The switch Multiple-Selection Structure

```
switch ( integer expression )
{
    case constant1 :
        statement(s)
        break ;
    case constant2 :
        statement(s)
        break ;

    . . .
    default: :
        statement(s)
        break ;
}
```

Notes:

- `break` and `default` are keywords
- If no `break`, execution flows through to next case
- If no `default`, switch might not do execute anything

switch Example

```
switch ( day ) {  
    case 1: System.out.println ( "Monday\n" );  
        break ;  
    case 2: System.out.println ( "Tuesday\n" );  
        break ;  
    case 3: System.out.println ( "Wednesday\n" );  
        break ;  
    case 4: System.out.println ( "Thursday\n" );  
        break ;  
    case 5: System.out.println ( "Friday\n" );  
        break ;  
    case 0:  
    case 6: System.out.println ( "Weekend\n" );  
        break ;  
    default: System.out.println ( "Error -- invalid day.\n" );  
        break ;  
}
```

Variable Scope

Variable scope:

- That set of code statements in which the variable is known to the compiler
- Where it can be referenced in your program.
- Limited to the ***code block*** in which it is defined.
 - A ***code block*** is a set of code enclosed in braces (***{ }***).

One interesting application of this principle allowed in Java involves the **for loop** construct.

for-loop index

- Can declare and initialize variables in the heading of a **for loop**.
- These variables are local to the for-loop.
- They may be reused in other loops.

```
String s = "hello world";  
int count = 1;  
for (int i = 0; i < s.length(); i++)  
{  
    count *= 2;  
}  
//using 'i' here generates a compiler error
```

Named Constants

- No “hard coded” values inside code!
- Declare constants as ***named constants***, and use their name instead

```
public static final int INCHES_PER_FOOT = 12;  
public static final double RATE = 0.14;
```

- The “**final**” modifier prevents a value from being changed inadvertently.
- More about **public** and **static** later
- Naming convention for constants
 - Use all uppercase letters
 - Designate word boundaries with an underscore character

Comments and Documentation

Comments

- **Line comment**

- Begins with the symbols `//`
- Compiler ignores remainder of the line
- Used for the coder or for a programmer who modifies the code

```
if (birthYear > currentYear) // birth year is invalid
    then . . .
```

- **Block comment**

- Begins with `/*` and ends with `*/`
- Compiler ignores anything in between
- Can span several lines
- Provides documentation for the users of the program

```
/* File: Date
   Author: Joe Smith
   Date: 9/1/09
  */
```

Comments & Named Constants

Display 1.8 Comments and a Named Constant

```
1  /**
2   Program to show interest on a sample account balance.
3   Author: Jane Q. Programmer.
4   E-mail Address: janeq@somemachine.etc.etc.
5   Last Changed: September 21, 2004.
6  */
7  public class ShowInterest
8  {
9      public static final double INTEREST_RATE = 2.5;
10
11     public static void main(String[] args)
12     {
13         double balance = 100;
14         double interest; //as a percent
15
16         interest = balance * (INTEREST_RATE/100.0);
17         System.out.println("On a balance of $" + balance);
18         System.out.println("you will earn interest of $"
19                             + interest);
20         System.out.println("All in just one short year.");
21     }
22 }
```

Although it would not be as clear, it is legal to place the definition of INTEREST_RATE here instead.

SAMPLE DIALOGUE

On a balance of \$100.0
you will earn interest of \$2.5
All in just one short year.

Special Javadoc Comment Form

- Similar to block comment, but:
 - Begins with `/**`
 - Not special to Java: considered same as `/*`
 - Processed by separate Javadoc program that creates HTML documentation pages from program source
 - Known set of embedded tags have special meaning to Javadoc.
 - E.g.: `@param`, `@return`
 - For an example:
<http://download.oracle.com/javase/6/docs/api/java/lang/String.html>

Strings

The **String** Class

- No primitive type for strings in Java
- **String** is a predefined class in the Java language.
 - Used to store *and process* strings
- Objects of type **String** are made up of strings of characters within double quotes.
 - Any quoted string is a constant of type **String**.

"Live long and prosper."

- A variable (object) of type **String** can be given the value of a **String** constant.

```
String blessing = "Live long and prosper."  
String greeting = "Hello";  
String name = "Bob";
```

String Concatenation

- Use the + operator

```
String greeting = "Hello";  
String name = "Bob";  
greeting + name is equal to "HelloBob"
```

- Any number of strings can be concatenated together.
- When a string is combined with almost any other type of item, the result is a string

```
"The answer is " + 42 evaluates to  
"The answer is 42"
```

- Strings also support the += operator

```
String greeting = "Hello";  
greeting += " Bob"; changes greeting to "Hello Bob"
```

String Methods

- The **String** class contains many useful **methods** (operations) for string-processing applications.
- Calling a **String** method:

String-object-name.method-name (arguments); OR

variable = String-object-name.method-name (arguments);

- Example

```
String greeting = "Hello";  //greeting is an object
int count = greeting.length();
System.out.println("Length is " + greeting.length());
```

Some Methods in the Class `String` (1 of 4)

Display 1.4 Some Methods in the Class `String`

`int` `length()`

Returns the length of the calling object (which is a string) as a value of type `int`.

EXAMPLE

After program executes `String greeting = "Hello!";`
`greeting.length()` returns 6.

`boolean` `equals(Other_String)`

Returns `true` if the calling object string and the *Other_String* are equal. Otherwise, returns `false`.

EXAMPLE

After program executes `String greeting = "Hello";`
`greeting.equals("Hello")` returns `true`
`greeting.equals("Good-Bye")` returns `false`
`greeting.equals("hello")` returns `false`

Note that case matters. "Hello" and "hello" are not equal because one starts with an uppercase letter and the other starts with a lowercase letter.

(continued)

Some Methods in the Class `String` (2 of 4)

Display 1.4 Some Methods in the Class `String`

`boolean` `equalsIgnoreCase(Other_String)`

Returns `true` if the calling object string and the *Other_String* are equal, considering uppercase and lowercase versions of a letter to be the same. Otherwise, returns `false`.

EXAMPLE

After program executes `String name = "mary!";`
`greeting.equalsIgnoreCase("Mary!")` returns `true`

`String` `toLowerCase()`

Returns a string with the same characters as the calling object string, but with all letter characters converted to lowercase.

EXAMPLE

After program executes `String greeting = "Hi Mary!";`
`greeting.toLowerCase()` returns `"hi mary!"`.

(continued)

Some Methods in the Class `String` (3 of 4)

Display 1.4 Some Methods in the Class `String`

`String toUpperCase()`

Returns a string with the same characters as the calling object string, but with all letter characters converted to uppercase.

EXAMPLE

After program executes `String greeting = "Hi Mary!";`
`greeting.toUpperCase()` returns `"HI MARY!"`.

`String trim()`

Returns a string with the same characters as the calling object string, but with leading and trailing white space removed. Whitespace characters are the characters that print as white space on paper, such as the blank (space) character, the tab character, and the new-line character `'\n'`.

EXAMPLE

After program executes `String pause = " Hmm ";`
`pause.trim()` returns `"Hmm"`.

(continued)

Some Methods in the Class `String` (4 of 4)

Display 1.4 Some Methods in the Class `String`

`char` `charAt(Position)`

Returns the character in the calling object string at the *Position*. Positions are counted 0, 1, 2, etc.

EXAMPLE

After program executes `String greeting = "Hello!";`
`greeting.charAt(0)` returns 'H', and
`greeting.charAt(1)` returns 'e'.

`String` `substring(Start)`

Returns the substring of the calling object string starting from *Start* through to the end of the calling object. Positions are counted 0, 1, 2, etc. Be sure to notice that the character at position *Start* is included in the value returned.

EXAMPLE

After program executes `String sample = "AbcdefG";`
`sample.substring(2)` returns "cdefG".

(continued)

Escape Sequences

Display 1.6 Escape Sequences

`\"` Double quote.

`\'` Single quote.

`\\` Backslash.

`\n` New line. Go to the beginning of the next line.

`\r` Carriage return. Go to the beginning of the current line.

`\t` Tab. White space up to the next tab stop.

-
- The character following the backslash does not have its usual meaning.
 - It is formed using two symbols, but regarded as a single character.

Pitfall: Using == with Strings

- The equality operator (==) can test the stored values of two values of a *primitive* type.

```
int x = 5, y = 5;  
if (x == y) . . .           // returns true
```

- When applied to two *objects*, == tests to see if they are stored in the *same memory location*. Example:

```
String string1 = "hello";  
String string2 = "hello";  
if (string1 == string2) . . . // returns false
```

- To test two strings to see if they have equal *values*, use the `String` method `equals`, or `equalsIgnoreCase`.

```
if (string1.equals(string2))           // returns true  
    or  
if (string1.equalsIgnoreCase(string2)) // returns true
```

Other Pitfalls with Strings

- Be careful with concatenation: associativity and promotion still applies:
 - Consider the following two expressions:
`4 + 2 + "is the answer to everything";`
VS.:
`"The answer to everything is " + 4 + 2;`
- A String is immutable
 - There is no way to modify any chars in a String:
 - E.g.: `"someString.charAt(x)"` doesn't let you change that char
 - But what does "immutable" really mean? Consider:
`String immutable = "Yes";`
`immutable = "No";`
`// Why is this allowed? And what of "+="?`
(See bad example)

Arrays

Arrays

- **Array**: A data structure used to process a collection of data that is all of the same type.
- An array is declared and created using the **new** operator.

```
BaseType[] ArrayName = new BaseType[size];
```

- The **size** may be given
 - as a non-negative integer, or
 - as an expression that evaluates to a nonnegative integer.

```
char[] line = new char[80];
```

```
double[] reading = new double[count];
```


Declaring vs. Creating Arrays

- Example

```
double[] score = new double[5];
```

or, using two statements:

```
double[] score;           // declares  
score = new double[5];    // creates
```

- The 1st statement declares **score** to be of the array type **double[]** (an array of doubles).
- The 2nd statement
 - creates an array with five numbered values of type **double**
 - makes the variable **score** a name for the array

The `length` Instance Variable

- An array is considered to be an object.
- Every array has exactly one *instance variable* (characteristic) named `length`.
 - When an array is created, the instance variable `length` is automatically set equal to its *size*.
 - The value of `length` cannot be changed (other than by creating an entirely new array using `new`).

```
double[] score = new double[5];
```

- Given `score` above, `score.length` has a value of 5.

Initializing Arrays

- An array can be initialized when it is declared.
- Example:

```
int[] age = {2, 12, 1};
```

- Given `age` above, `age.length` automatically has a value of 3.

```
System.out.print("Length is " + age.length);  
prints  
Length is 3
```

Notes on Arrays

- index starts at 0.
- arrays can't shrink or grow.
- each element is initialized.
- array bounds checking (no overflow!)
 - `ArrayIndexOutOfBoundsException`

Initializing Arrays

- Using a **for** loop,

```
double[] reading = new double[100];  
for(int index = 0; index < reading.length; index++) {  
    reading[index] = 42.0;  
}
```

- Using array literals:

```
int[] foo = {1,2,3,4,5};  
String[] names = {"Joe", "Sam"};
```

- If the elements of an array are not initialized explicitly, they will automatically be initialized to the default value for their **base type**.

An Array Coding Exercise

- Write a code fragment that finds the smallest value in an array of integers.

Arrays as Parameters

- An array may be a method argument. Example:

```
public void doubleElements(double[] a)    // a = address
{
    for (int i = 0; i < a.length; i++)    // notice use
        a[i] = a[i]*2;                    // of a.length
}
```

- Given arrays of `double` as follows:

```
double[] a = new double[10];
double[] b = new double[30];
```

the method `doubleElements` can be invoked as follows:

```
doubleElements(a);
doubleElements(b);
```

Pitfall: Use of = with Arrays

- An array variable contains the *memory address* of the array it names.
- The assignment operator (=) only copies this memory address.

```
int a[ ] = {1, 2, 3};  
int b[ ] = new int[3];  
  
b = a;    // b and a are now names for  
          // the same array
```


Pitfall: Use of = with Arrays

- A **for** loop is usually used to make two different arrays have the same values in each indexed position.

```
int i;  
int a[ ] = {1, 2, 3};  
int b[ ] = new int[3];  
for (i = 0; (i < a.length) && (i < b.length); i++)  
    b[i] = a[i];
```

- Note that the above code will not make **b** an exact copy of **a**, unless **a** and **b** have the same length

Pitfall: Use of == with Arrays

- The equality operator (==) only tests two arrays to see if they are stored in the same *memory location*.

(a == b)

is **true** if **a** and **b** reference the *same array*.

Otherwise, it is **false**.

- An **equalsArray** method can be defined to test arrays for *value equality*.
 - The following method tests two integer arrays to see if they contain the same integer values.

Code to Test for Value Equality

```
public boolean equalsArray(int[] a, int[] b)
{
    if (a.length == b.length)
    {
        int i = 0;
        boolean elementsMatch = true;
        while (i < a.length && elementsMatch)
        {
            if (a[i] != b[i])
                elementsMatch = false;
            i++;
        }
        return elementsMatch;
    }
    else
        return false;
}
```

Strings and Arrays Are Objects

- It's important to keep in mind that despite syntactic shortcuts (e.g., “hello” + “bye”, foo[x]), strings and arrays *are* objects
 - They have real methods
 - They have constructors, which must be called to create new instances.
 - Otherwise, you just have null references.

Exception Handling

Exceptions

- Terminology:
 - *throw an exception*: signal that some condition (possibly an error) has occurred.
 - *catch an exception*: deal with the error (or whatever).
- In Java, exception handling is necessary (forced by the compiler)!

Try/Catch/Finally

```
try {  
    // code that can throw an exception  
} catch (ExceptionType1 e1) {  
    // code to handle the exception  
} catch (ExceptionType2 e2) {  
    // code to handle the exception  
} catch (Exception e) {  
    // code to handle other exceptions  
} finally {  
    // code to run after try or any catch  
}
```

Exception Handling

- Exceptions take care of handling errors
 - instead of returning an error, some method calls will throw an exception.
- Can be dealt with at any point in the method invocation stack.
- Forces the programmer to be aware of what errors can occur and to deal with them.

Exception Example

```
static String squareNumberString(String str) {  
    int n;  
    try {  
        n = Integer.parseInt(str);  
    } catch (NumberFormatException e) {  
        System.err.println("Error: invalid integer \"" + str + "\"");  
        System.exit(1);  
    }  
    return "\"" + Math.pow(n,2);  
}
```

A Better Exception Example

```
static String squareNumberString(String str) {  
    int n;  
    try {  
        n = Integer.parseInt(str);  
    } catch (NumberFormatException e) {  
        throw new IllegalArgumentException("str must contain a  
            valid integer");  
    }  
    return "" + Math.pow(n,2);  
}
```

Input/Output

- The java.io package provides classes for reading and writing streaming (sequential) data
- Example: reading lines from the console

```
import java.io.*;
```

```
BufferedReader console = new BufferedReader(new  
                                           InputStreamReader(System.in));
```

```
System.out.print("Enter your name: ");
```

```
String name = null;
```

```
try {
```

```
    name = console.readLine();
```

```
} catch (IOException e) {
```

```
    System.err.println("Fatal input error: "+e);
```

```
    System.exit(1);
```

```
}
```

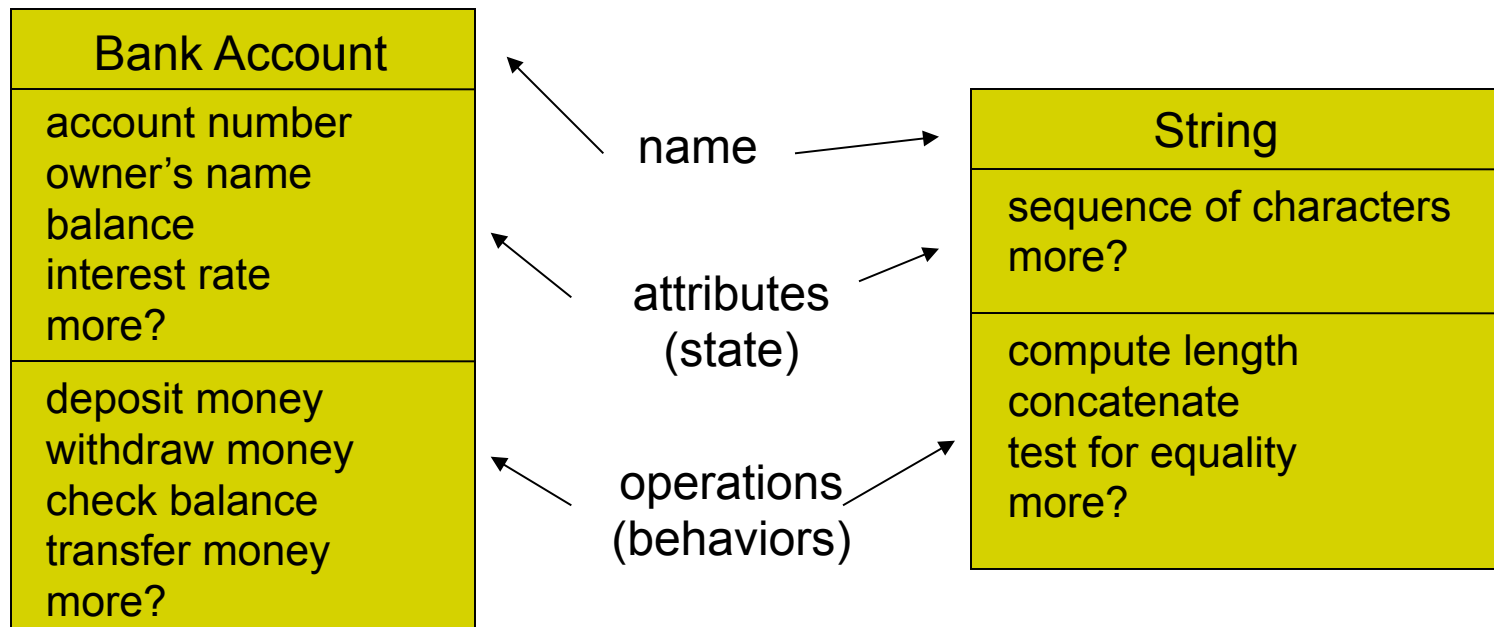
```
System.out.println("Hello "+name);
```

- File input/output is similar, but more on that later

Objects and Classes

What's an Object?

- Must first define a **class**
 - A data type containing
 - Attributes - make up the object's "state"
 - Operations - define the object's "behaviors"



So, an object is ...

- a particular “instance” of a class.

Berg's Account

12-345-6
Jen Berg
\$1,250.86
1.5%

Frede's Account

65-432-1
Dennis Frede
\$5.50
2.7%

Mitchell's Account

43-261-5
Sarah Mitchell
\$825.50
2.5%

For any of these accounts, one can

- deposit money
- withdraw money
- check the balance
- transfer money

Class Definitions

- You already know
 - how to use classes and the objects created from them, and
 - how to invoke their methods.
- For example, you have already been using the predefined **String** class.

```
String name = "Fido";  
System.out.println("name length = " + name.length  
());
```

A Class Is a Type

- A class is a programmer-defined type.
- Variables can be declared of a class type.
- A value of a class variable type is called an ***object*** or an ***instance*** of the class.
 - If A is a class, then the phrases
 - “X is of type A”
 - “X is an object of the class A”
 - “X is an instance of the class A”

mean the same thing

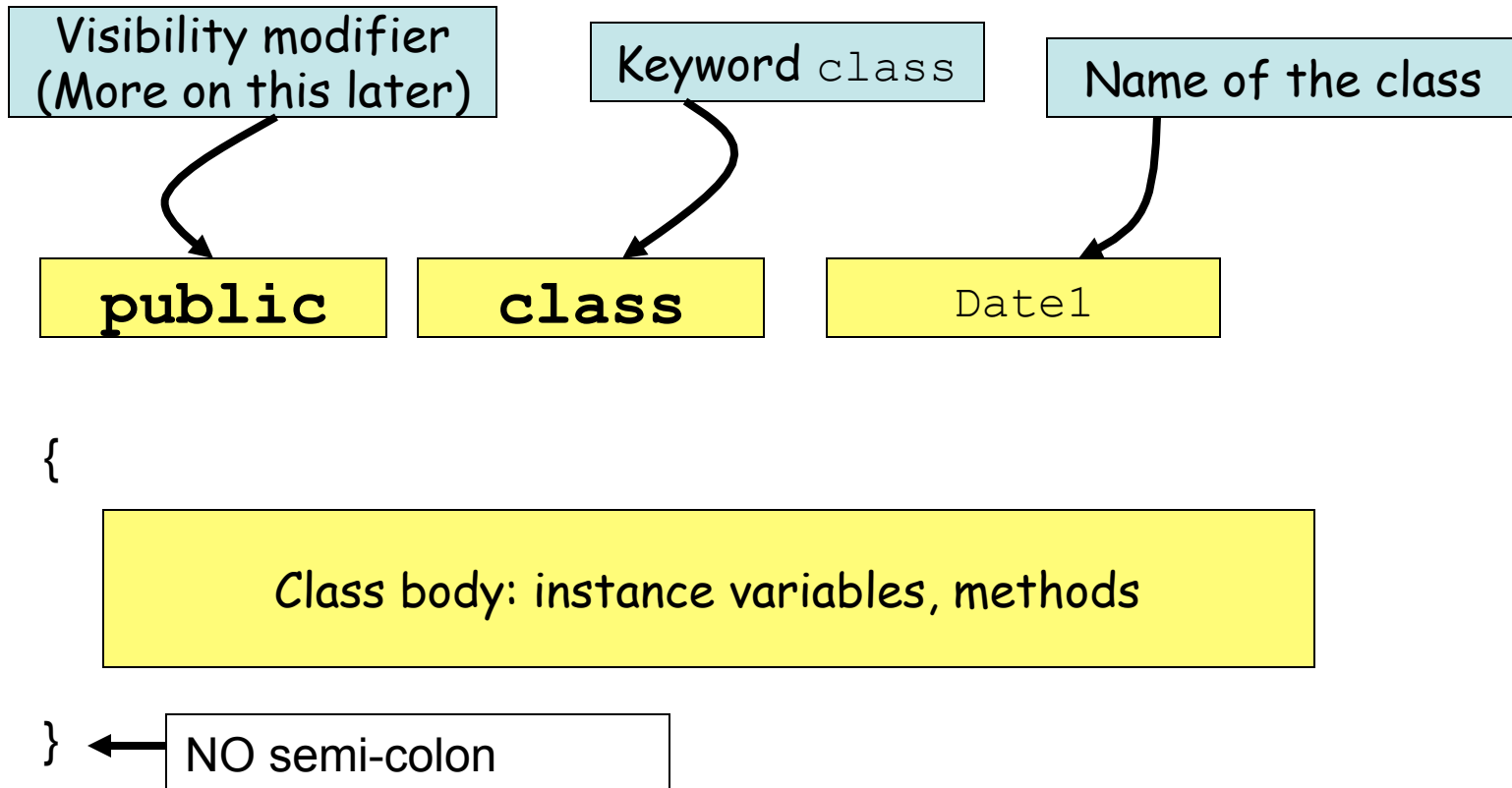
Objects

- All objects of a class have the same methods.
- All objects of a class have the same attributes (i.e., name, type, and number).
 - For different objects, each attribute can hold a different value.
 - The values of the attributes define the object ***state***, which is what makes each object unique.

The Class Definition

- A ***class definition*** implements the class model.
 - The class behaviors/services/actions/operations are implemented by class ***methods***.
 - The class attributes (data items) are called ***fields*** or ***instance variables***.
- In Java, classes are defined in files with the .java extension.
- The name of the file must match the name of the class defined within it.
 - e.g. class ‘Baker’ must be in Baker.java

Anatomy of a Java Class

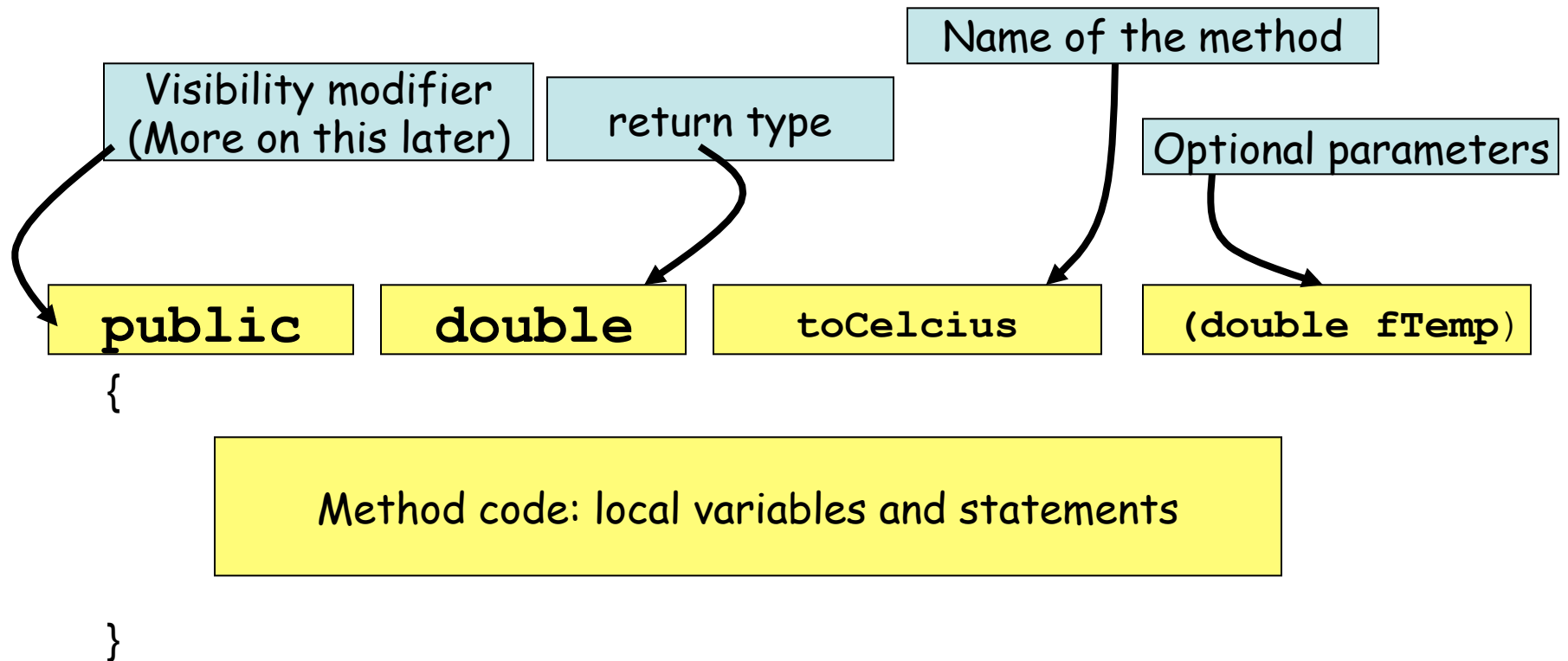


Instance Variables

- Defined inside the class definition
- May be
 - primitive types
 - other class types
- Are accessible by all methods of the class
 - have ***class scope***
- Given the services identified for the red-green-yellow traffic light, the garage door opener and the bank account, what instance variables might be defined for each?

Anatomy of a Method

Are very much like functions



Example: A Date Class

This class definition goes in a file named **Date1.java**.

```
public class Date1  
{
```

```
    public String month;  
    public int day;  
    public int year;
```

These are the (public) “data members” or “instance variables” of the class

```
    public String toString( )  
    {  
        return month + “ “ + day + “ , “ + year;  
    }  
}
```

This is a method definition and its implementation

A method may use the class instance variables

Date1 toString Method

- **toString** is a method of the Date1 class.
 - Its definition and implementation are part of the Date1 class.
- Class methods may
 - be void or return a value, and
 - (optionally) have parameters, which may be
 - primitive types passed by value, and/or
 - objects (discussed later).
- All of a class' methods have access to all of the class' instance variables (class scope).

Using Date1

This class definition goes in a file named [Date1Demo.java](#).

```
public class Date1Demo
{
```

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

```
        Date1 myDate;
        myDate = new Date1( );
```

Create a Date1 object
named myDate

```
        myDate.month = "July";
        myDate.day = 4;
        myDate.year = 2007;
```

Give values to the data
members

```
        String dateString = myDate.toString( );
        System.out.println(dateString);
```

Invoke the toString method

```
    }
```

```
}
```


Creating the Date1 Object

- The statement `Date1 myDate;` defines a variable of type Date1.
 - But there is no Date1 object yet!
- The statement `myDate = new Date1();` creates a “new” Date1 object and names it with the variable “myDate”.
 - Now “myDate” ***refers to*** a Date1 object.
- For convenience, these statements can be combined.

```
Date1 myDate = new Date1( );
```

“Dot” Notation

- Public instance variables of an object are referenced using the “dot” operator.

```
myDate.month = "July";  
myDate.day = 4;  
myDate.year = 2011;
```

- Instance variables can be used like any other variable of the same type.
- The set of values stored in all instance variables define the ***state*** of the myDate object.

More “Dot” Motation

- The statement

```
myDate.toString( );
```

invokes the `toString` method of `myDate`, which refers to an object of type `Date`1.

- In OO terminology, we say that we are “sending the `toString` message” to the object referred to by `myDate`.
- The object `myDate` is referred to as the ***calling object*** or ***host object***.

Other Date Methods

Some other possible services that the Date1 class might provide:

- incrementDay - changes the date to “tomorrow”
- DMYString – creates a different string format
- setDate - initialize/change the year, month, and/or day
- What others ?

New Date1 Methods

```
// change the month (using an int), day, and year.
public void setDate( int newMonth, int newDay, int newYear )
{
    month = monthString( newMonth );
    day = newDay;
    year = newYear;
}

// change month number (int) to string - used by setDate
public String monthString( int monthNumber ) {
    switch ( monthNumber ) {
        case 1: return "January";
        case 2: return "February";
        case 3: return "March";
        case 4: return "April";
        case 5: return "May";
        case 6: return "June";
        case 7: return "July";
        case 8: return "August";
        case 9: return "September";
        case 10: return "October";
        case 11: return "November";
        case 12: return "December";
        default: return "????";
    }
}
```

Confusion?

- In the preceding **setDate** method it's tempting to define the method using the common terms “month”, “day” and “year” as the parameters.

```
public void setDate( int month, int day, int year)
{
    month = monthString( month ); // which month is which?
    day = day;                    // which day is which?
    year = year;                  // which year is which?
}
```

The compiler assumes that all uses of **day**, **month**, and **year** refer to the *method parameters* and hence this code has no effect.

Calling Object

When any class method is called, the instance variables used within the method are assumed to belong to the calling/host object.

What the code in **setDate** is really trying to do is

```
public void setDate( int month, int day, int year)
{
    "calling object".month = monthString( month );
    "calling object".day = day;
    "calling object".year = year;
}
```

It's handy (and sometimes necessary) to have a name for the calling object.

In Java, we use the reserved word **this** as the generic name of the calling object.

Using **this**

So, if we want to name our parameters the same as our instance variables:

```
public void setDate( int month, int day, int year)
{
    this.month = monthString( month );    // notice "this"
    this.day = day;
    this.year = year;
}
```

Note:

- Many examples in the text use this technique for class methods.
- Some Java programmer tools (including Eclipse) use this technique when writing code for you.

this Again

Recall the `toString` method from `Date1`:

```
public void toString( )
{
    return month + " " + day + " " + year;
}
```

It's clear that `month`, `day`, and `year` refer to the instance variables of the calling object because there are no parameters.

We could have written:

```
public void toString( )
{
    return this.month + " " + this.day + " " + this.year;
}
```

If the prefix `this` is unnecessary, it is usually omitted.

Sample Code Segment Using Date1

```
Date1 newYears = new Date1( );  
newYears.month = "January";  
newYears.day = 1;  
newYears.year = 2011;
```

```
Date1 birthday = new Date1( );  
birthday.month = "July";  
birthday.day = 4;  
birthday.year = 1776;
```

```
System.out.println(newYears.toString( ));           // line 1  
System.out.println(birthday.toString( ));           // line 2  
System.out.println(birthday.monthString(6));        // line 3  
birthday.setDate( 2, 2, 2002);                      // line 4  
System.out.println(birthday.toString( ));           // line 5  
newYears.day = 42;                                    // line 6  
System.out.println(newYears.toString( ));           // line 7
```

August 42, 2011

- It appears that classes allow the user to change the data anytime he or she chooses, possibly making the data invalid.
- That's true so far because we have defined our instance variables with **public** access.
- This is rarely the case in real applications.

More About Methods

- Different classes can define a method with the same name.
- Java can determine which method to call based on the type of the calling object.
- Example:

```
Date1 birthday = new Date1( );  
Dog fido = new Dog( );  
System.out.println(birthday.toString( ));  
System.out.println(fido.toString( ));
```

- `birthday.toString()` will call the `toString()` method defined in the `Date1` class because `birthday`'s type is `Date1`.
- `fido.toString()` will call the `toString()` method defined in the `Dog` class because `fido`'s type is `Dog`.

Method Overloading

- Two or more methods *in the same class* may also have the same name.
- This technique is known as ***method overloading***.

Overloaded setDate

- The Date1 class `setDate` method:

```
public boolean setDate( int month, int day, int year )
```

- Suppose we wanted to change only the day and year?
 - Define another method named setDate:

```
public boolean setDate( int day, int year )
```

(After all, setDate is a good descriptive name for what this method does.)

Date2 Class - Overloaded setDate Method

```
public class Date2
{
    public String month;
    public int day;           // 1 - 31
    public int year;          // 4 digits

    public boolean setDate( int newMonth, int newDay, int newYear )
    {
        // code here
    }

    public boolean setDate( int newDay, int newYear );
    {
        // code here, doesn't change month
    }

    // toString( ), monthString( ), etc. follow
}
```

Date2Demo Class

```
public class Date2Demo
{
    public static void main (String[ ] args)
    {
        Date2 myDate = new Date2( );

        myDate.setDate( 1, 23, 1982 );
        System.out.println( myDate.toString( ) );
        myDate.setDate( 4, 1999 );
        System.out.println( myDate.toString( ) );
    }
}
```

How does Java know which setDate method to invoke?

Method Signature

- A method is uniquely identified by
 - its name and
 - its parameter list (parameter types and their order).
- This is known as its ***signature***.

Examples:

```
public boolean setDate(int newMonth, int newDay, int newYear)
public boolean setDate(String newMonth, int newDay, int newYear)
public boolean setDate(int newDay, int newYear)
public boolean setDate(int newDay, String newMonth)
```

Return Type is Not Enough

- Suppose we attempt to create an overloaded `setDay()` method by using different return types.

```
public void setDay( int day )      { /* code here */ }  
public boolean setDay( int day ) { /* code here */ }
```

- This is NOT valid method overloading because the code that calls `setDay()` can ignore the return value.

```
birthday.setDay( 22 );
```

- The compiler can't tell which `setDay()` method to invoke.
- Just because a method returns a value doesn't mean the caller has to use it.

Too Much of a Good Thing

Automatic type promotion and overloading can sometimes interact in ways that confuse the compiler. Example:

```
public class X {  
    //version 1  
    public void printAverage ( int a, double b) {  
        /*code*/  
    }  
  
    //version 2  
    public void printAverage ( double a, int b) {  
        /*code*/  
    }  
}
```

Why might this be problematic?

Too Much of a Good Thing

```
public void printAverage ( int a, double b) { /*code*/ }  
public void printAverage ( double a, int b) { /*code*/ }
```

- Now, consider this:

```
X myX = new X( );  
myX.printAverage( 5, 7 );
```

- The Java compiler can't decide whether to:
 - promote 7 to 7.0 and invoke the first version of printAverage(), or
 - promote 5 to 5.0 and invoke the second.
- It will throw up its hands and complain
- Take-home lesson: don't be too clever with method overloading

More Documentation

Class-level Documentation

- Class header format:

```
/**  
 * File: Table.java  
 * Project: CMSC 206 Assignment 1, Fall 2011  
 * Date: 9/29/2011  
 * E-mail: jdoe22@brynmawr.edu  
 * Class Description:  
 * @author Jane Doe  
 */
```

Method-level Documentation

- Method header format:

```
/**
 * Name: circleArea
 * PreCondition: the radius is greater than zero
 * PostCondition: none
 * @param radius - the radius of the circle
 * @return the calculated area of the circle
 * (@throws - optional)
 */
double circleArea ( double radius ) {
    // handle unmet precondition
    if (radius < 0.0) {
        return 0.0;
    } else {
        return Math.PI * radius * radius;
    }
}
```

Instance Variable Documentation

- Javadoc wants the variable descriptions on line before actual declaration:

```
/** first name of the account holder */
String firstName;
/**
 * the last name of the account holder
 * (note we can have a multi-line description).
 */
String lastName;
```


Method Documentation

- Clear communication with the class user is of paramount importance so that he can
 - use the appropriate method, and
 - use class methods properly.
- Method comments:
 - explain what the method does, and
 - describe how to use the method.
- Two important types of method comments:
 - ***precondition*** comments
 - ***post-conditions*** comments

Preconditions and Postconditions

- **Precondition**
 - What is assumed to be true when a method is called
 - If any pre-condition is not met, the method may not correctly perform its function.
- **Postcondition**
 - States what will be true after the method executes (assuming all pre-conditions are met)
 - Describes the side-effect of the method, e.g. if state of instance changes

An Example

Very often the precondition specifies the limits of the parameters and the postcondition says something about the return value.

```
/*
  Pre-condition:
    1 <= month <= 12
    day appropriate for the month
    1000 <= year <= 9999
  Post-condition:
    The month, day, and year of the calling object
    have been set to the parameter values.
  @return true if the calling object has been changed,
    false otherwise
*/
public boolean setDate(int month, int day, int year)
{
    // code here
}
```