

Cmp Sci 187: Introduction to Java

Based on Appendix A of text
(Koffmann and Wolfgang)

Topics of the Review

- Essentials of *object-oriented programming, in Java*
- Java primitive data types, control structures, and arrays
- Using some predefined classes:
 - **Math**
 - **JOptionPane**, I/O streams
 - **String**, **StringBuffer**, **StringBuilder**
 - **StringTokenizer**
- Writing *and documenting* your own Java classes

Some Salient Characteristics of Java

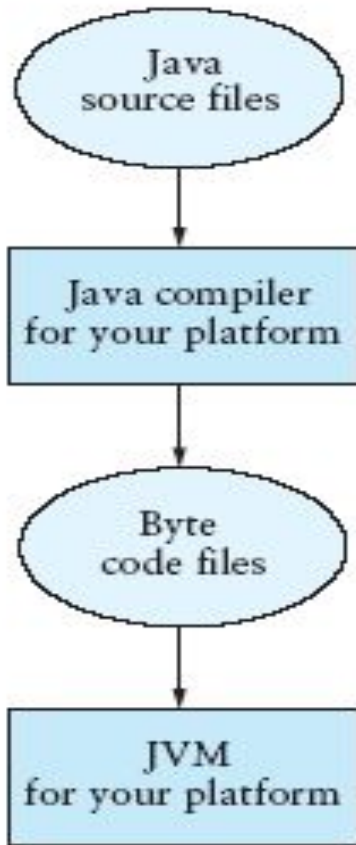
- Java is ***platform independent***: the same program can run on any correctly implemented Java system
- Java is ***object-oriented***:
 - Structured in terms of ***classes***, which group data with operations on that data
 - Can construct new classes by ***extending*** existing ones
- Java designed as
 - A ***core language*** plus
 - A rich collection of ***commonly available packages***
- Java can be embedded in Web pages

Java Processing and Execution

- Begin with Java **source code** in text files: **Model.java**
- A Java source code compiler produces Java **byte code**
 - Outputs one file per class: **Model.class**
 - May be standalone or part of an IDE
- A **Java Virtual Machine** loads and executes class files
 - May compile them to native code (e.g., x86) internally

Compiling and Executing a Java Program

FIGURE A.1
Compiling and Executing
a Java Program



Classes and Objects

- The **class** is the unit of programming
- A Java program is a **collection of classes**
 - Each class definition (usually) in its own `.java` file
 - *The file name must match the class name*
- A class describes **objects (instances)**
 - Describes their common characteristics: is a *blueprint*
 - Thus all the instances have these same characteristics
- These characteristics are:
 - **Data fields** for each object
 - **Methods** (operations) that do work on the objects

Grouping Classes: The Java API

- API = *Application Programming Interface*
- Java = small core + extensive collection of packages
- A **package** consists of some related Java classes:
 - Swing: a GUI (graphical user interface) package
 - AWT: Application Window Toolkit (more GUI)
 - util: utility data structures (important to CS 187!)
- The ***import*** statement tells the compiler to make available classes and methods of another package
- A ***main*** method indicates where to begin executing a class (if it is designed to be run as a program)

A Little Example of `import` and `main`

```
import javax.swing.*;
    // all classes from javax.swing
public class HelloWorld { // starts a class
    public static void main (String[] args) {
        // starts a main method
        // in: array of String; out: none (void)
    }
}
```

- `public` = can be seen from any package
- `static` = not “part of” an object

Processing and Running `HelloWorld`

- `javac HelloWorld.java`
 - Produces `HelloWorld.class` (byte code)
- `java HelloWorld`
 - Starts the JVM and runs the `main` method

References and Primitive Data Types

- Java distinguishes two kinds of entities
 - Primitive types
 - Objects
- Primitive-type data is stored in primitive-type variables
- Reference variables store the *address of* an object
 - No notion of “object (physically) in the stack”
 - No notion of “object (physically) within an object”

Primitive Data Types

- Represent numbers, characters, boolean values
- Integers: byte, short, int, and long
- Real numbers: float and double
- Characters: char

Primitive Data Types

Data type	Range of values
byte	-128 .. 127 (8 bits)
short	-32,768 .. 32,767 (16 bits)
int	-2,147,483,648 .. 2,147,483,647 (32 bits)
long	-9,223,372,036,854,775,808 (64 bits)
float	+/-10 ⁻³⁸ to +/-10 ⁺³⁸ and 0, about 6 digits precision
double	+/-10 ⁻³⁰⁸ to +/-10 ⁺³⁰⁸ and 0, about 15 digits precision
char	Unicode characters (generally 16 bits per char)
boolean	True or false

Primitive Data Types (continued)

TABLE A.2

The First 128 Unicode Symbols

	000	001	002	003	004	005	006	007
0	Null		Space	0	0	P	'	p
1			!	1	A	Q	a	q
2			"	2	B	R	b	r
3			#	3	C	S	c	s
4			\$	4	D	T	d	t
5			%	5	E	U	e	u
6			&	6	F	V	f	v
7	Bell		'	7	G	W	g	w
8	Backspace		(8	H	X	h	x
9	Tab)	9	I	Y	I	y
A	Line feed		*	:	J	Z	j	z
B		Escape	+	:	K	[k	{
C	Form feed		,	<	L	\	l	
D	Return		-	=	N]	m	}
E			.	>	N	^	n	~
F			/	?	0	_	o	delete

Operators

1. subscript `[]`, call `()`, member access `.`
2. pre/post-increment `++` `--`, boolean complement `!`, bitwise complement `~`, unary `+` `-`, type cast `(type)`, object creation `new`
3. `*` `/` `%`
4. binary `+` `-` (`+` also concatenates strings)
5. signed shift `<<` `>>`, unsigned shift `>>>`
6. comparison `<` `<=` `>` `>=`, class test `instanceof`
7. equality comparison `==` `!=`
8. bitwise and `&`
9. bitwise or `|`

Operators

11. logical (sequential) and `&&`
12. logical (sequential) or `||`
13. conditional `cond ? true-expr : false-expr`
14. assignment `=`, compound assignment `+= -= *= /=`
`<<= >>= >>>= &= |=`

Type Compatibility and Conversion

- **Widening conversion:**
 - In operations on mixed-type operands, the numeric type of the smaller range is converted to the numeric type of the larger range
 - In an assignment, a numeric type of smaller range can be assigned to a numeric type of larger range
- `byte` to `short` to `int` to `long`
- `int` kind to `float` to `double`

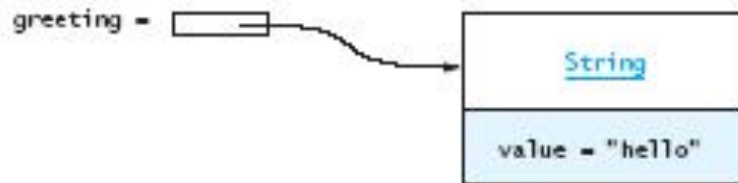
Declaring and Setting Variables

- `int square;`
`square = n * n;`
- `double cube = n * (double)square;`
 - Can generally declare local variables where they are initialized
 - All variables get a safe initial value anyway (zero/null)

Referencing and Creating Objects

- You can **declare reference variables**
 - They reference objects of **specified types**
- Two reference variables can reference **the same object**
- The **new** operator creates an instance of a class
- A **constructor** executes when a new object is created
- Example: **String greeting = "hello";**

FIGURE A.2
Variable greeting
References a String
Object



Java Control Statements

- A group of statements executed in order is written
 - `{ stmt1; stmt2; ...; stmtN; }`
- The statements execute in the order 1, 2, ..., N
- Control statements alter this sequential flow of execution

Java Control Statements (continued)

TABLE A.4

Java Control Statements

Control Structure	Purpose	Syntax
if ... else	Used to write a decision with <i>conditions</i> that select the alternative to be executed. Executes the first (second) alternative if the <i>condition</i> is true (false).	<pre>if (<i>condition</i>) { ... } else { ... }</pre>
switch	Used to write a decision with scalar values (integers, characters) that select the alternative to be executed. Executes the <i>statements</i> following the <i>label</i> that is the <i>selector</i> value. Execution falls through to the next case if there is no return or break . Executes the statements following default if the <i>selector</i> value does not match any <i>label</i> .	<pre>switch (<i>selector</i>) { case <i>label</i> : <i>statements</i>; break; case <i>label</i> : <i>statements</i>; break; ... default : <i>statements</i>; }</pre>
while	Used to write a loop that specifies the repetition <i>condition</i> in the loop header. The <i>condition</i> is tested before each iteration of the loop and, if it is true, the loop body executes; otherwise, the loop is exited.	<pre>while (<i>condition</i>) { ... }</pre>
for	Used to write a loop that specifies the <i>initialization</i> , repetition <i>condition</i> , and <i>update</i> steps in the loop header. The <i>initialization</i> statements execute before loop repetition begins, the <i>condition</i> is tested before each iteration of the loop and, if it is true, the loop body executes; otherwise, the loop is exited. The <i>update</i> statements execute after each iteration.	<pre>for (<i>initialization</i>; <i>condition</i>; <i>update</i>) { ... }</pre>

Java Control Statements (continued)

TABLE A.4 (continued)

Control Structure	Purpose	Syntax
<code>do ... while</code>	Used to write a loop that specifies the repetition <i>condition</i> after the loop body. The <i>condition</i> is tested after each iteration of the loop and, if it is true, the loop body is repeated; otherwise, the loop is exited. The loop body always executes at least one time.	<pre>do { ... while (<i>condition</i>) ;</pre>

Methods

- A Java method defines a group of statements as performing a particular operation
- **static** indicates a **static** or **class** method
- A method that is not **static** is an **instance** method
- All method arguments are **call-by-value**
 - Primitive type: *value* is passed to the method
 - Method may modify local copy **but** will not affect caller's value
 - Object reference: *address of object* is passed
 - Change to reference variable does not affect caller
 - **But** operations can affect the object, visible to caller

The Class Math

TABLE A.5
Class Math Methods

Method	Behavior
static <i>numeric</i> abs(<i>numeric</i>)	Returns the absolute value of its <i>numeric</i> argument (the result type is the same as the argument type).
static double ceil(double)	Returns the smallest whole number that is not less than its argument.
static double cos(double)	Returns the trigonometric cosine of its argument (an angle in radians).
static double exp(double)	Returns the exponential number <i>e</i> (i.e., 2.718 ...) raised to the power of its argument.
static double floor(double)	Returns the largest whole number that is not greater than its argument.
static double log(double)	Returns the natural logarithm of its argument.
static <i>numeric</i> max(<i>numeric</i> , <i>numeric</i>)	Returns the larger of its <i>numeric</i> arguments (the result type is the same as the argument types).
static <i>numeric</i> min(<i>numeric</i> , <i>numeric</i>)	Returns the smaller of its <i>numeric</i> arguments (the result type is the same as the argument type).
static double pow(double, double)	Returns the value of the first argument raised to the power of the second argument.
static double random()	Returns a random number greater than or equal to 0.0 and less than 1.0.
static double rint(double)	Returns the closest whole number to its argument.
static long round(double)	Returns the closest long to its argument.
static int round(float)	Returns the closest int to its argument.
static double sin(double)	Returns the trigonometric sine of its argument (an angle in radians).
static double sqrt(double)	Returns the square root of its argument.
static double tan(double)	Returns the trigonometric tangent of its argument (an angle in radians).
static double toDegrees(double)	Converts its argument (in radians) to degrees.
static double toRadians(double)	Converts its argument (in degrees) to radians.

Escape Sequences

- An escape sequence is a sequence of two characters beginning with the character \
- A way to represents special characters/symbols

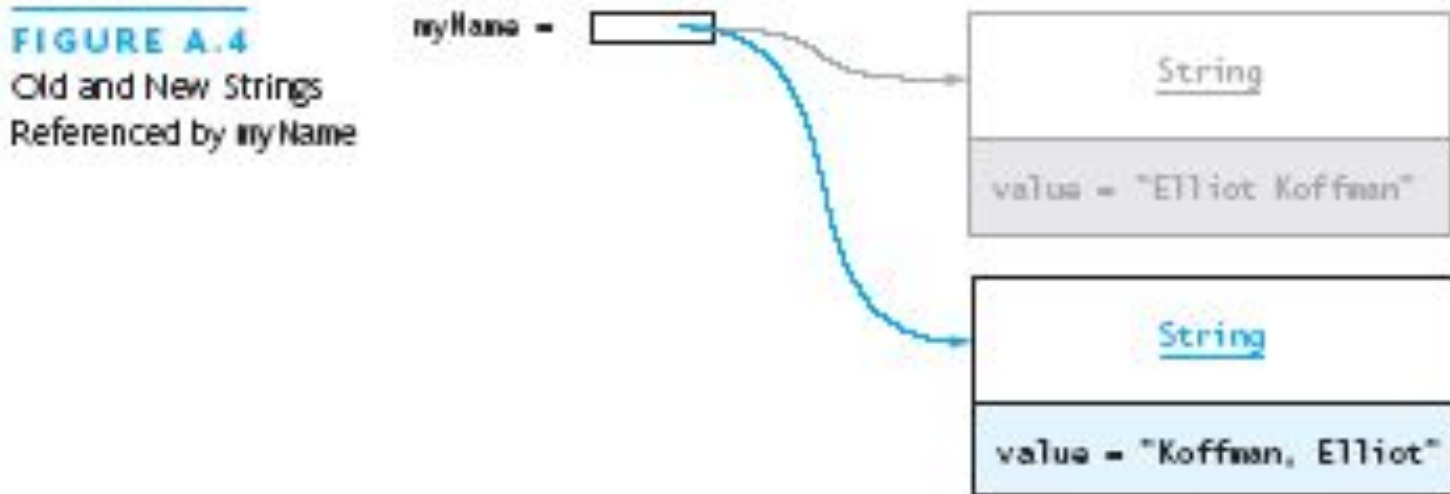
TABLE A.6

Escape Sequences

Sequence	Meaning
<code>\n</code>	Start a new output line
<code>\t</code>	Tab character
<code>\\</code>	Backslash character
<code>\"</code>	Double quote
<code>\'</code>	Single quote or apostrophe
<code>\u$dddd$</code>	The Unicode character whose code is $dddd$ where each digit d is a hexadecimal digit in the range 0 to F (0-9, A-F)

The `String` Class

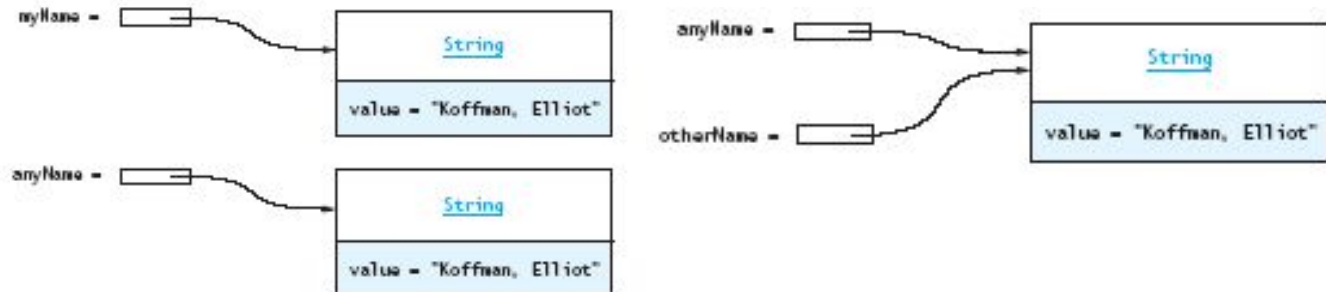
- The `String` class defines a data type that is used to store a sequence of characters
- You cannot modify a `String` object
 - If you attempt to do so, Java will create a new object that contains the modified character sequence



Comparing Objects

- You **can't use the relational or equality operators** to compare the values stored in strings (or other objects)
(You will compare the *pointers*, not the *objects*!)

FIGURE A.5
Two String Objects at
Different Addresses
with the Same Contents



The StringBuffer Class

- Stores character sequences
- Unlike a **String** object, you **can** change the contents of a **StringBuffer** object

TABLE A.8

StringBuffer Methods in java.lang.StringBuffer

Method	Behavior
<code>void StringBuffer append(<i>anyType</i>)</code>	Appends the string representation of the argument to this StringBuffer. The argument can be of any data type.
<code>int capacity()</code>	Returns the current capacity of this StringBuffer.
<code>void StringBuffer delete(int start, int end)</code>	Removes the characters in a substring of this StringBuffer, starting at position <code>start</code> and ending with the character at position <code>end - 1</code> .
<code>void StringBuffer insert(int offset, <i>anyType</i> data)</code>	Inserts the argument data (any data type) into this StringBuffer at position <code>offset</code> , shifting the characters that started at <code>offset</code> to the right.
<code>int length()</code>	Returns the length (character count) of this StringBuffer.
<code>StringBuffer replace(int start, int end, String str)</code>	Replaces the characters in a substring of this StringBuffer (from position <code>start</code> through position <code>end - 1</code>) with characters in the argument <code>str</code> . Returns this StringBuffer.
<code>String substring(int start)</code>	Returns a new string containing the substring that begins at the specified index <code>start</code> and extends to the end of this StringBuffer.
<code>String substring(int start, int end)</code>	Return a new string containing the substring in this StringBuffer from position <code>start</code> through position <code>end - 1</code> .
<code>String toString()</code>	Returns a new string that contains the same characters as this StringBuffer object.

StringTokenizer Class

- We often need to process individual pieces, or *tokens*, of a **String**

TABLE A.9

StringTokenizer Methods in java.util.StringTokenizer

Method	Behavior
StringTokenizer(String str)	Constructs a new StringTokenizer object for the string specified by str. The delimiters are "whitespace" characters (space, newline, tab, and so on).
StringTokenizer(String str, String delim)	Constructs a new StringTokenizer object for the string specified by str. The delimiters are the characters specified in delim.
boolean hasMoreTokens()	Returns true if this tokenizer's string has more tokens; otherwise, returns false.
String nextToken()	Returns the next token of this tokenizer's string if there is one; otherwise, a run-time error will occur.

Wrapper Classes for Primitive Types

- Sometimes we need to process primitive-type data as objects
- Java provides a set of classes called wrapper classes whose objects contain primitive-type values: **Float**, **Double**, **Integer**, **Boolean**, **Character**, etc.

TABLE A.10

Methods for Class Integer

Method	Behavior
<code>int compareTo(Integer anInt)</code>	Compares two Integers numerically.
<code>double doubleValue()</code>	Returns the value of this Integer as a double.
<code>boolean equals(Object obj)</code>	Returns true if the value of this Integer is equal to its argument's value; returns false otherwise.
<code>int intValue()</code>	Returns the value of this Integer as an int.
<code>static int parseInt(String s)</code>	Parses the string argument as a signed integer.
<code>String toString()</code>	Returns a String object representing this Integer's value.

Defining Your Own Classes

- *Unified Modeling Language* (UML) is a standard diagram notation for describing a class

FIGURE A.6
Class Diagram for
Person

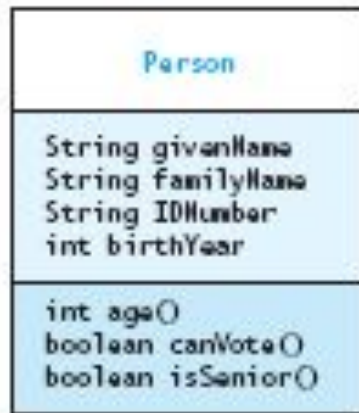
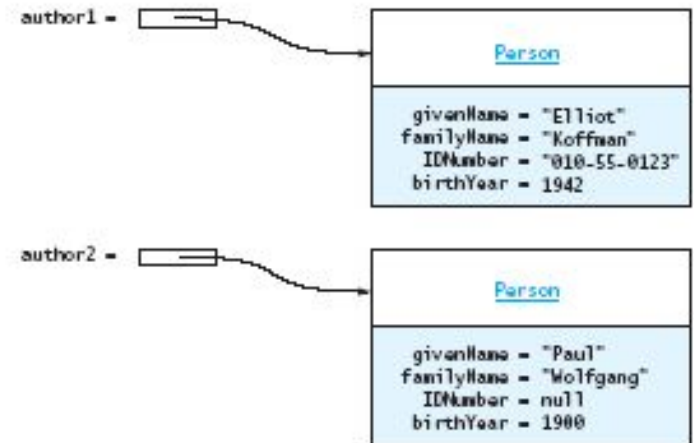


FIGURE A.7
Object Diagrams of
Two Instances of Class
Person



Field
signatures:
type and name

Method *signatures*:
name, argument
types, result type

Class
name

Field
values

Class
name

Defining Your Own Classes (continued)

- The modifier `private` limits access to just this class
- Only class members with `public` visibility can be accessed outside of the class* (* but see `protected`)
- **Constructors** initialize the data fields of an instance

TABLE A.11

Default Values for Data Fields

Data Field Type	Default Value
<code>int</code> (or other integer type)	0
<code>double</code> (or other real type)	0.0
<code>boolean</code>	<code>false</code>
<code>char</code>	<code>\u0000</code> (the smallest Unicode character: the null character)
Any reference type	<code>null</code>

The Person Class

```
// we have omitted javadoc to save space
public class Person {
    private String givenName;
    private String familyName;
    private String IDNumber;
    private int birthYear;

    private static final int VOTE_AGE = 18;
    private static final int SENIOR_AGE = 65;
    ...
}
```


The Person Class (2)

```
// constructors: fill in new objects
public Person(String first, String family,
               String ID, int birth) {
    this.givenName    = first;
    this.familyName  = family;
    this.IDNumber     = ID;
    this.birthYear   = birth;
}
public Person (String ID) {
    this.IDNumber = ID;
}
```

The Person Class (3)

```
// modifier and accessor for givenName
public void setGivenName (String given) {
    this.givenName = given;
}

public String getGivenName () {
    return this.givenName;
}
```

The Person Class (4)

```
// more interesting methods ...
public int age (int inYear) {
    return inYear - birthYear;
}
public boolean canVote (int inYear) {
    int theAge = age(inYear);
    return theAge >= VOTE_AGE;
}
```

The Person Class (5)

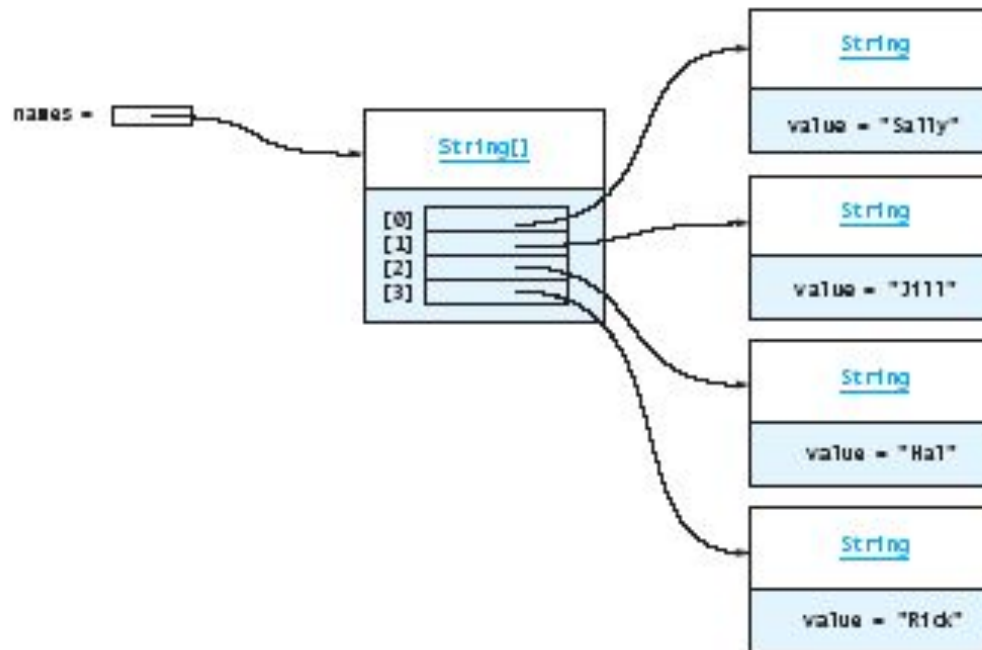
```
// "printing" a Person
public String toString () {
    return "Given name: " + givenName + "\n"
        + "Family name: " + familyName + "\n"
        + "ID number: " + IDNumber + "\n"
        + "Year of birth: " + birthYear + "\n";
}
```

The Person Class (6)

```
// same Person?  
public boolean equals (Person per) {  
    return (per == null) ? false :  
        this.IDNumber.equals(per.IDNumber);  
}
```

Arrays

- In Java, an array is also an object
- The elements are indexes and are referenced using the form **arrayvar[subscript]**



Array Example

```
float grades[] = new float[numStudents];  
... grades[student] = something; ...  
  
float total = 0.0;  
for (int i = 0; i < grades.length; ++i) {  
    total += grades[i];  
}  
System.out.printf("Average = %6.2f%n",  
                  total / numStudents);
```

Array Example Variations

```
// possibly more efficient
for (int i = grades.length; --i >= 0; ) {
    total += grades[i];
}
```

```
// uses Java 5.0 "for each" looping
for (float grade : grades) {
    total += grade;
}
```


Input/Output using Class `JOptionPane`

- Java 1.2 and higher provide class `JOptionPane`, which facilitates display
 - Dialog windows for input
 - Message windows for output

Input/Output using Class `JOptionPane` (continued)

TABLE A.13

Methods from Class `JOptionPane`

Method	Behavior
<code>static String showInputDialog(String prompt)</code>	Displays a dialog window that displays the argument as a prompt and returns the character sequence typed by the user.
<code>static void showMessageDialog(Object parent, String message)</code>	Displays a window containing a message string (the second argument) inside the specified container (the first argument).

FIGURE A.15

A Dialog Window (Left)
and Message Window
(Right)



Converting Numeric Strings to Numbers

- A dialog window always returns a reference to a **String**
- Therefore, a conversion is required, using **static** methods of class **String**:

TABLE A.14

Methods for Converting Strings to Numbers

Method	Behavior
<code>static int parseInt(String)</code>	Returns an <code>int</code> value corresponding to its argument string. A <code>NumberFormatException</code> occurs if its argument string contains characters other than digits.
<code>static double parseDouble(String)</code>	Returns a <code>double</code> value corresponding to its argument string. A <code>NumberFormatException</code> occurs if its argument string does not represent a real number.

Input/Output using Streams

- An **InputStream** is a sequence of characters representing program input data
- An **OutputStream** is a sequence of characters representing program output
- The console keyboard stream is **System.in**
- The console window is associated with **System.out**

Opening and Using Files: Reading Input

```
import java.io.*;
public static void main (String[] args) {
    // open an input stream    (**exceptions!)
    BufferedReader rdr =
        new BufferedReader(
            new FileReader(args[0]));
    // read a line of input
    String line = rdr.readLine();
    // see if at end of file
    if (line == null) { ... }
```

Opening and Using Files: Reading Input (2)

```
// using input with StringTokenizer  
StringTokenizer sTok =  
    new StringTokenizer (line);  
while (sTok.hasMoreElements()) {  
    String token = sTok.nextToken();  
    ...;  
}  
  
// when done, always close a stream/reader  
rdr.close();
```

Alternate Ways to Split a **String**

- Use the `split` method of `String`:

```
String[] = s.split("\\s");
```

```
// see class Pattern in java.util.regex
```

- Use a `StreamTokenizer` (in `java.io`)

Opening and Using Files: Writing Output

```
// open a print stream    (**exceptions!)
PrintStream ps = new PrintStream(args[0]);
// ways to write output
ps.print("Hello");    // a string
ps.print(i+3);        // an integer
ps.println(" and goodbye.");    // with NL
ps.printf("%2d %12d%n", i, 1<<i); // like C
ps.format("%2d %12d%n", i, 1<<i); // same
// closing output streams is very important!
ps.close();
```


Summary of the Review

- A Java program is a collection of classes
- The JVM approach enables a Java program written on one machine to execute on any other machine that has a JVM
- Java defines a set of primitive data types that are used to represent numbers, characters, and boolean data
- The control structures of Java are similar to those found in other languages
- The Java **String** and **StringBuffer** classes are used to reference objects that store character strings

Chapter Review (continued)

- Be sure to use methods such as `equals` and `compareTo` to compare the *contents* of `String` objects
- You can declare your own Java classes and create objects of these classes using the `new` operator
- A class has data fields and instance methods
- Array variables can reference array objects
- Class `JOptionPane` can be used to display dialog windows for data entry and message windows for output
- The stream classes in package `java.io` read strings from the console and display strings to the console, and also support file I/O