Introduction to Java

Aliya Farooq

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

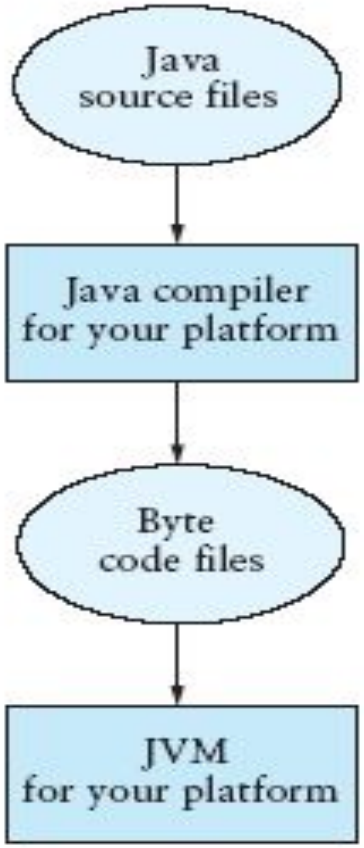
- Java is ***platform independent:*** the same program can run on any correctly implemented Java system
- Java is ***object-oriented:***
  - Structured in terms of ***classes***, which group data with operations on that data
  - Can construct new classes by ***extending*** existing ones
- Java designed 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

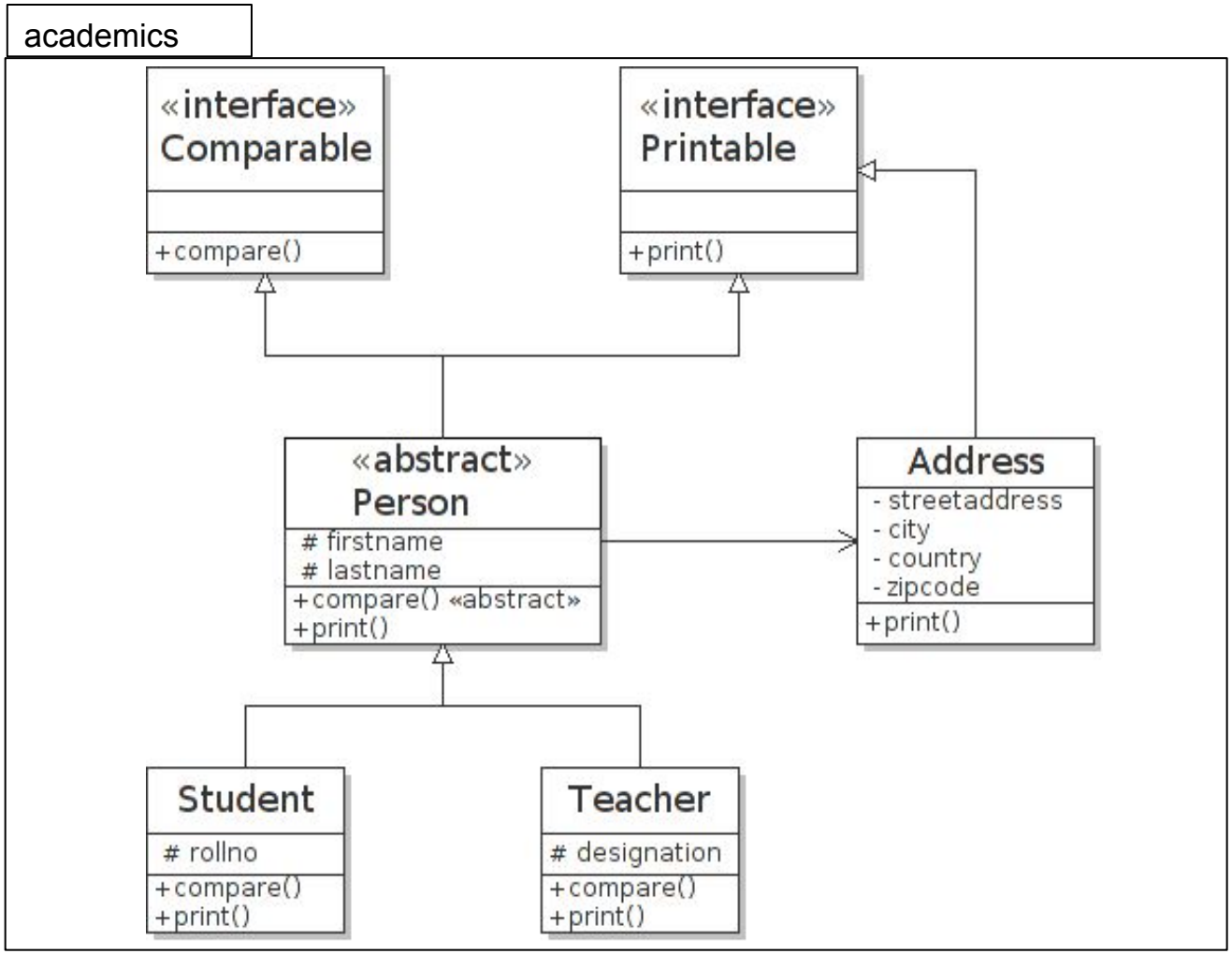


Classes and Objects

- 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

- Java has:
  - same syntax as C++ with minor variations
  - Complete support for Object-oriented programming
  - Garbage collection
- Java does not have:
  - Pointers
    - Instead **references** exist for manipulating objects
  - Multiple inheritance
    - Instead multiple **interfaces** can be implemented
  - Templates
    - Instead **generics** are used for generic programming using principles of polymorphism

Example


academics
«interface»
Comparable
+compare()
«interface»
Printable
+print()
«abstract»
Person
firstname
lastname
+compare() «abstract»
+print()
Address
- streetaddress
- city
- country
- zipcode
+print()
Student
rollno
+compare()
+print()
Teacher
designation
+compare()
+print()

Interface

- Interface
  - Purely virtual / abstract
  - Only contains method signatures
  - Java 8 allows default methods for evolving interfaces
  - Used to define a public interface for a class
  - Class implements an interface
  - Class can implement multiple interfaces leading to multiple type inheritance

Comparable.java

```
package academics;

public interface Comparable {
 int compare(Comparable obj) throws NotComparableException;
}
```

Printable.java

```
package academics;

public interface Printable {
 void print();
}
```

Abstract class

- Abstract class
  - cannot be instantiated
  - at least one abstract method
  - abstract keyword required with class and method
  - may implement interfaces
  - not providing the implementation of an interface method automatically makes the class abstract

Person.java

```
package academics;

public abstract class Person implements Comparable, Printable{

 protected String firstName;
 protected String lastName;
 protected Address address;

 public Person(){

 }

 public Person(String fname,String lname){
  firstName = fname;
  lastName = lname;
 }

 public String getName(){
  return firstName + " " + lastName;
 }


 public void print(){
  System.out.println(getName());

  try{
   address.print();
  }
  catch(NullPointerException ex){
   System.out.println("No address recorded");
  }

  System.out.println("---");
 }

 public abstract int compare(Comparable obj) throws NotComparableException;
}
```

Address.java

```
package academics;

public class Address implements Printable{

 private String streetAddress;
 private String city;
 private String country;
 private String zipCode;

 @Override
 public void print() {
  System.out.println(streetAddress + "\n" +
                     city       + "\n" +
                     country              + "\n" +
                     zipCode              + "\n" );
 }

}
```

Inheritance

- Inheritance
  - specified through **extends** keyword
  - only single inheritance allowed – cannot inherit from multiple classes
  - Overriding a method hides the parent implementation but can be accessed through **super** keyword
  - super may also be used in constructor for initialization of parent – must be the first statement in constructor
  - all classes extend from Object by default

Student.java

```
package academics;

public class Student extends Person{

 protected String rollNumber;

 public Student(String rollNum,String fName,String lName){
  super(fName,lName);
  rollNumber = rollNum;
 }

 @Override
 public int compare(Comparable obj)
  throws NotComparableException{
   if (obj instanceof Student){
    Student student = (Student) obj;

    return this.getName().compareTo(student.getName());
  }
  else throw new NotComparableException();
 }

 @Override
 public void print() {
  System.out.println(rollNumber);
  super.print();
 }

}
```

Teacher.java

```
package academics;

public class Teacher extends Person{

 protected String designation;

 public Teacher(String designation,String fName,String lName){
  super(fName,lName);
  this.designation = designation;
 }

 @Override
 public int compare(Comparable obj)
  throws NotComparableException{
   if (obj instanceof Teacher){
    Teacher teacher = (Teacher) obj;

    return this.getName().compareTo(teacher.getName());
   }
   else throw new NotComparableException();
 }

 @Override
  public void print() {
   System.out.println(designation);
   super.print();
  }

}
```

Exception

- Exception
  - represents a runtime error or an exceptional situation that requires handling
  - Exception is expected in the **try** block and handled in the **catch** block – also use **finally** block for resource cleanup
  - A method either handles or throws an exception
  - An exception may be thrown using the **throw** keyword
  - Method signature must specify using **throws** keyword, the exceptions that may be thrown
  - A custom exception can be created by inheriting through **Exception** class

NotComparableException.java

```
package academics;

public class NotComparableException extends Exception{

}
```

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	@	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		-	=	M	]	m	}
E			.	>	N	^	n	~
F			/	?	O	_	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";`**

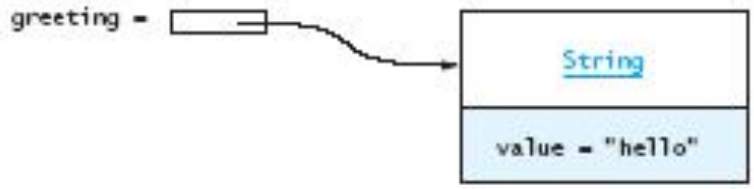


FIGURE A.2
Variable greeting References a String Object

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 *conditions* that select the alternative to be executed. Executes the first (second) alternative if the *condition* is true (false).	`if (condition) {`
		`  ...`
		`} else {`
		`  ...`
		`}`

`switch`	Used to write a decision with scalar values (integers, characters) that select the alternative to be executed. Executes the *statements* following the *label* that is the *selector* value. Execution falls through to the next `case` if there is no `return` or `break`. Executes the statements following `default` if the *selector* value does not match any *label*.	`switch (selector) {`
		`  case label : statements; break;`
		`  case label : statements; break;`
		`  ...`
		`  default : statements;`
		`}`

`while`	Used to write a loop that specifies the repetition *condition* in the loop header. The *condition* is tested before each iteration of the loop and, if it is true, the loop body executes; otherwise, the loop is exited.	`while (condition) {`
		`  ...`
		`}`

`for`	Used to write a loop that specifies the *initialization*, repetition *condition*, and *update* steps in the loop header. The *initialization* statements execute before loop repetition begins; the *condition* is tested before each iteration of the loop and, if it is true, the loop body executes; otherwise, the loop is exited. The *update* statements execute after each iteration.	`for (initialization; condition; update) {`
		`  ...`
		`}`

Java Control Statements (continued)

TABLE A.4 (continued)

Control Structure	Purpose	Syntax

`do ... while`	Used to write a loop that specifies the repetition *condition* after the loop body. The *condition* 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.	`do {`
		`  ...`
		`while (`*condition*`) ;`

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` *numeric* `abs(`*numeric*`)`	Returns the absolute value of its *numeric* 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 *e* (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` *numeric* `max(`*numeric*`,` *numeric*`)`	Returns the larger of its *numeric* arguments (the result type is the same as the argument types).
`static` *numeric* `min(`*numeric*`,` *numeric*`)`	Returns the smaller of its *numeric* 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
`\n`	Start a new output line
`\t`	Tab character
`\\`	Backslash character
`\"`	Double quote
`\'`	Single quote or apostrophe
`\u`*dddd*	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

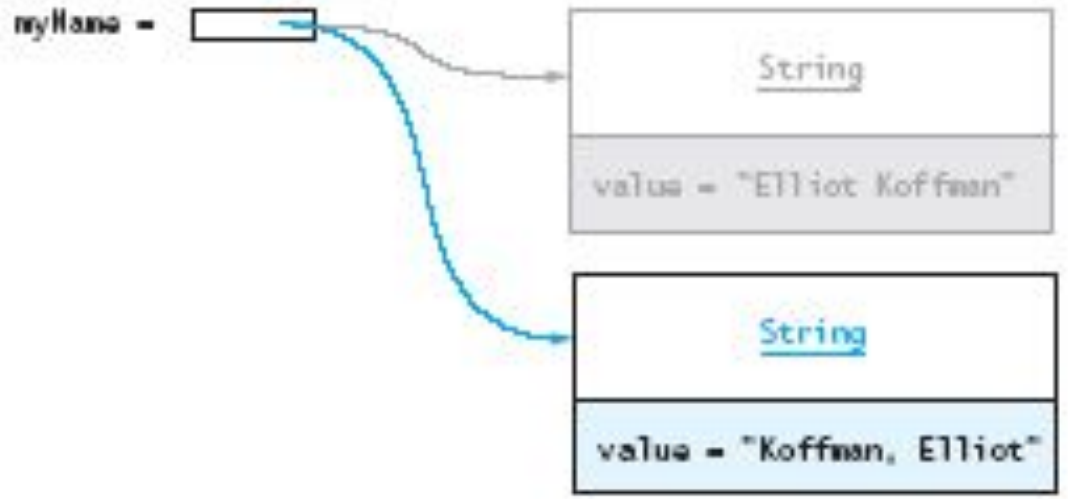


FIGURE A.4
Old and New Strings Referenced by myName

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*!)

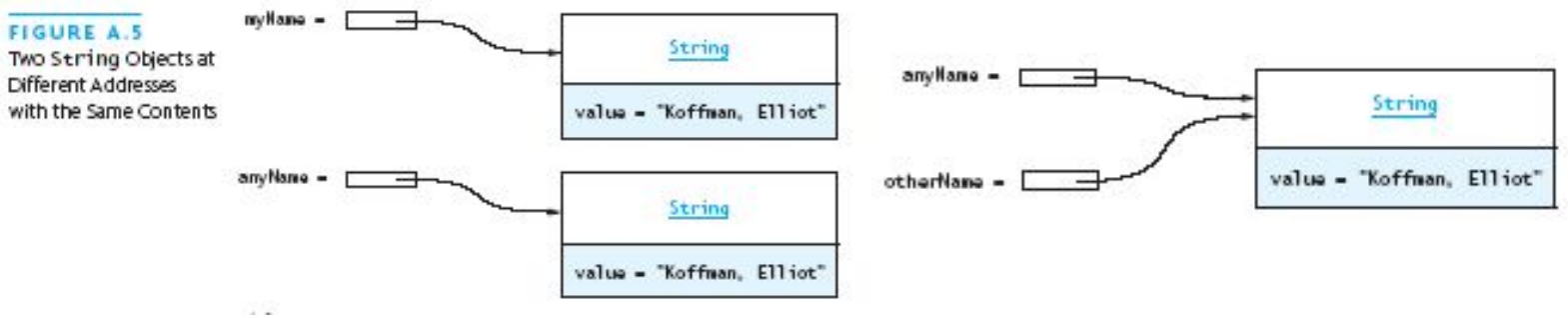


FIGURE A.5
Two String Objects at Different Addresses with the Same Contents

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

TABLE A.10
Methods for Class Integer

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

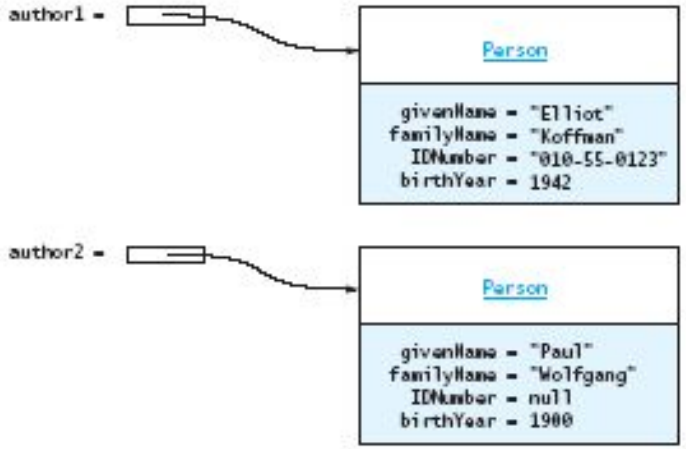
FIGURE A.6
Class Diagram for Person

Field *signatures*: type and name

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

Class name

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



Field *values*

Class name

Defining Your Own Classes (continued)

- The modifier **`private`** limits access to just this class
- Only class members with **`public`** visibility

TABLE A.11
Default Values for Data Fields

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

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)

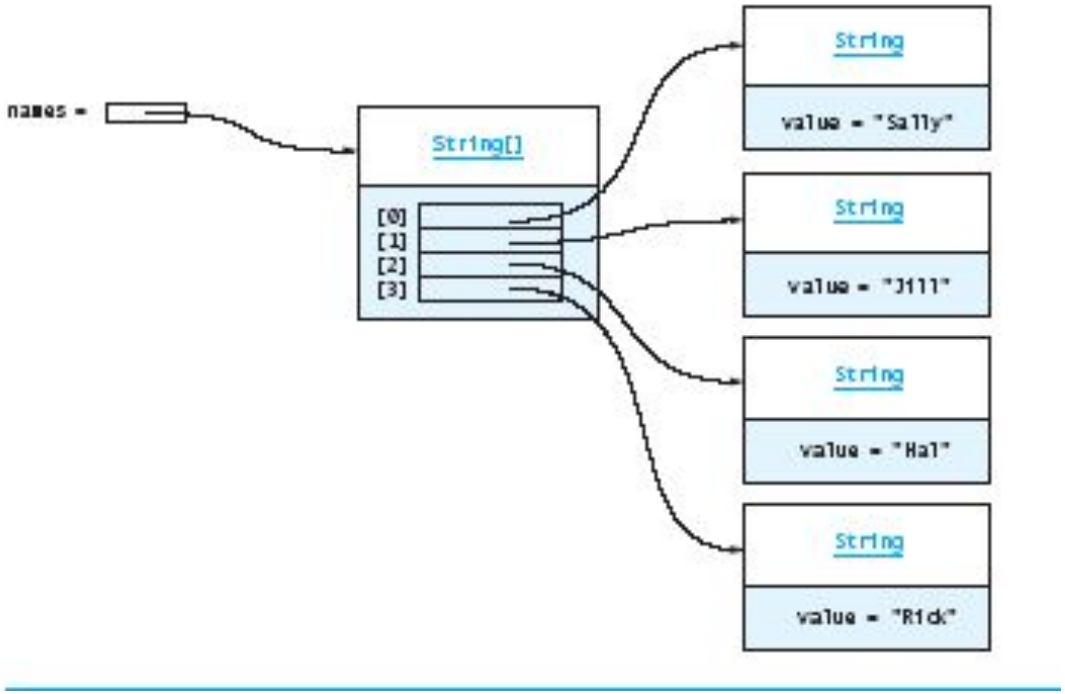
```
// “printing” a Person
public String toString () {
  return “Given name: “ + givenName +
  “\n”
   +   “Family name: “ + familyName +
  “\n”
   +   “ID number: “ + IDNumber + “\n”
   +   “Year of birth: “ + birthYear +
  “\n”;
}
```

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;
}
```

I/O Stream

- Available in java.io package
- Stream is a sequence of data
- I/O Stream represents an input source or an output destination
- Represents different sources
  - Examples: files, devices, other programs, etc
- Supports multiple formats
  - Byte Streams: represents data in bytes - extends from InputStream (or OutputStream)
  - Character Streams: represents data in characters according to Unicode conventions – descend from Reader (or Writer)
  - Buffered Streams: stores data in buffer during an I/O operation and read (or write) from (to) it until the buffer is empty

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
`static String showInputDialog(String prompt)`	Displays a dialog window that displays the argument as a prompt and returns the character sequence typed by the user.
`static void showMessageDialog(Object parent, String message)`	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`**

TABLE A.14
Methods for Converting Strings to Numbers

Method	Behavior
`static int parseInt(String)`	Returns an `int` value corresponding to its argument string. A `NumberFormatException` occurs if its argument string contains characters other than digits.
`static double parseDouble(String)`	Returns a `double` value corresponding to its argument string. A `NumberFormatException` 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

- 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

Summary

- 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