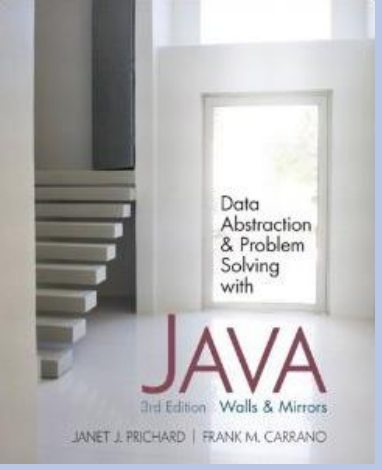


Chapter 1

Review of Java Fundamentals

Language Basics

- Java application
  - Collection of classes
    - One class contains the *main* method
- Java programs can also be written as applets

Comments

- Comment line
  - Begins with two slashes (//)
  - Continues until the end of the line
- Multiple-line comment
  - Begins with /* and ends with */
  - Useful for debugging
  - Cannot contain another multiple-line comment
- `javadoc` comments
  - Begins with /** and ends with */

Identifiers and Keywords

- Identifier
  - Sequence of letters, digits, underscores, and dollar signs
  - Must begin with either a letter or underscore
  - Cannot be a Java keyword
  - Used to name various parts of the program
  - Java distinguishes between uppercase and lowercase letters
- Keywords
  - Java reserved identifiers

Variables

- Represents a memory location
- Contains a value of primitive type or a reference
- Its name is a Java identifier
- Declared by preceding variable name with data type

```
double radius; // radius of a sphere
String name; // reference to a String object
```

Primitive Data Types

- Organized into four categories
  - Boolean
  - Character
  - Integer
  - Floating point
- Character and integer types are called integral types
- Integral and floating-point types are called arithmetic types

Primitive Data Types

Category	Data Type	Wrapper Class
Boolean	`boolean`	`Boolean`
Character	`char`	`Character`
Integer	`byte`	`Byte`
	`short`	`Short`
	`int`	`Integer`
	`long`	`Long`
Floating point	`float`	`Float`
	`double`	`Double`

Figure 1-5

Primitive data types and corresponding wrapper classes

Primitive Data Types

- Value of primitive type is not considered an object
- `java.lang` provides wrapper classes for each of the primitive types
- Autoboxing
  - Automatically converts from a primitive type to the equivalent wrapper class
- Auto-unboxing
  - Reverse process

References

- Data type used to locate an object
- Java does not allow programmer to perform operations on the reference value
- Location of object in memory can be assigned to a reference variable

Literal Constants

- Indicate particular values within a program
- Used to initialize the value of a variable
- Decimal integer constants
  - Do not use commas, decimal points, or leading zeros
  - Default data type is either `int` or `long`
- Floating constants
  - Written using decimal points
  - Default data type is `double`

Literal Constants

- Character constants
  - Enclosed in single quotes
  - Default data type is `char`
  - Literal character strings
    - Sequence of characters enclosed in double quotes

Named Constants

- Have values that do not change
- Declared as a variable but using the keyword `final`

Assignments and Expressions

- Expressions
  - Combination of variables, constants, operators, and parentheses
- Assignment statement
  - Example: `radius = r;`
- Arithmetic expression
  - Combine variables and constants with arithmetic operators and parentheses
    - Arithmetic operators: *, /, %, +, -

Assignments and Expressions

- Relational expressions
  - Combine variables and constants with relational, or comparison, and equality operators and parentheses
    - Relational or comparison operators: <, <=, >=. >
    - Equality operators: ==, !=
  - Evaluate to `true` or `false`

Assignments and Expressions

- Logical expressions
  - Combine variables and constants of arithmetic types, relational expressions with logical operators
    - Logical operators: &&, ||
  - Evaluate to `true` or `false`
  - Short-circuit evaluation
    - Evaluates logical expressions from left to right
    - Stops as soon as the value of expression is apparent

Assignments and Expressions

- Implicit type conversions
  - Occur during assignment and during expression evaluation
  - Right-hand side of assignment operator is converted to data type of item on left-hand side
  - Floating-point values are truncated not rounded
  - Integral promotion
    - Values of type `byte`, `char`, or `short` are converted to `int`
  - Conversion hierarchy
    - `int` → `long` → `float` → `double`

Assignments and Expressions

- Explicit type conversions
  - Possible by means of a cast
  - Cast operator
    - Unary operator
    - Formed by enclosing the desired data type within parentheses
- Multiple assignments
  - Embed assignment expressions within assignment expressions
    - Example: `a = 5 + (b = 4)`
    - Evaluates to `9` while `b` is assigned `4`

Assignments and Expressions

- Other assignment operators
  - -=
  - *=
  - /=
  - %=
  - ++
  - --

Arrays

- Collection of elements with the same data type
- Array elements have an order
- Support direct and random access
- One-dimensional arrays
  - Declaration example

```
final int DAYS_PER_WEEK = 7;
double [] maxTemps = new double[DAYS_PER_WEEK];
```

  - Length of an array is accessible using data field `length`
  - Use an index or subscript to access an array element

Arrays

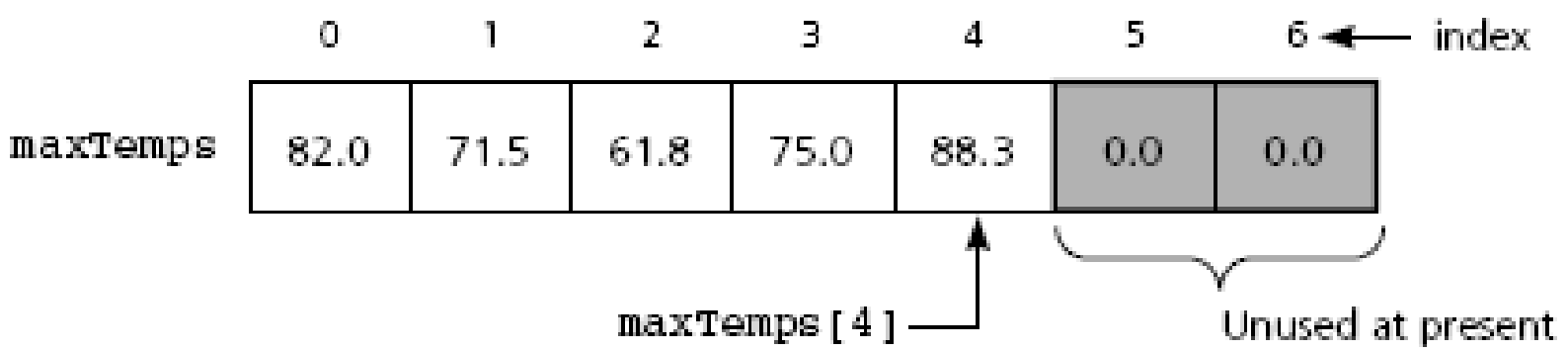


Figure 1-7

One-dimensional array of at most seven elements

Arrays

- One-dimensional arrays (continued)
  - Initializer list example

```
double [] weekDayTemps = {82.0, 71.5, 61.8,
  75.0, 88.3};
```

  - You can also declare array of object references
- Multidimensional arrays
  - Use more than one index
  - Declaration example

```
final int DAYS_PER_WEEK = 7;
final int WEEKS_PER_YEAR = 52;
double[][] minTemps = new
  double[DAYS_PER_WEEK][WEEKS_PER_YEAR];
```

Arrays

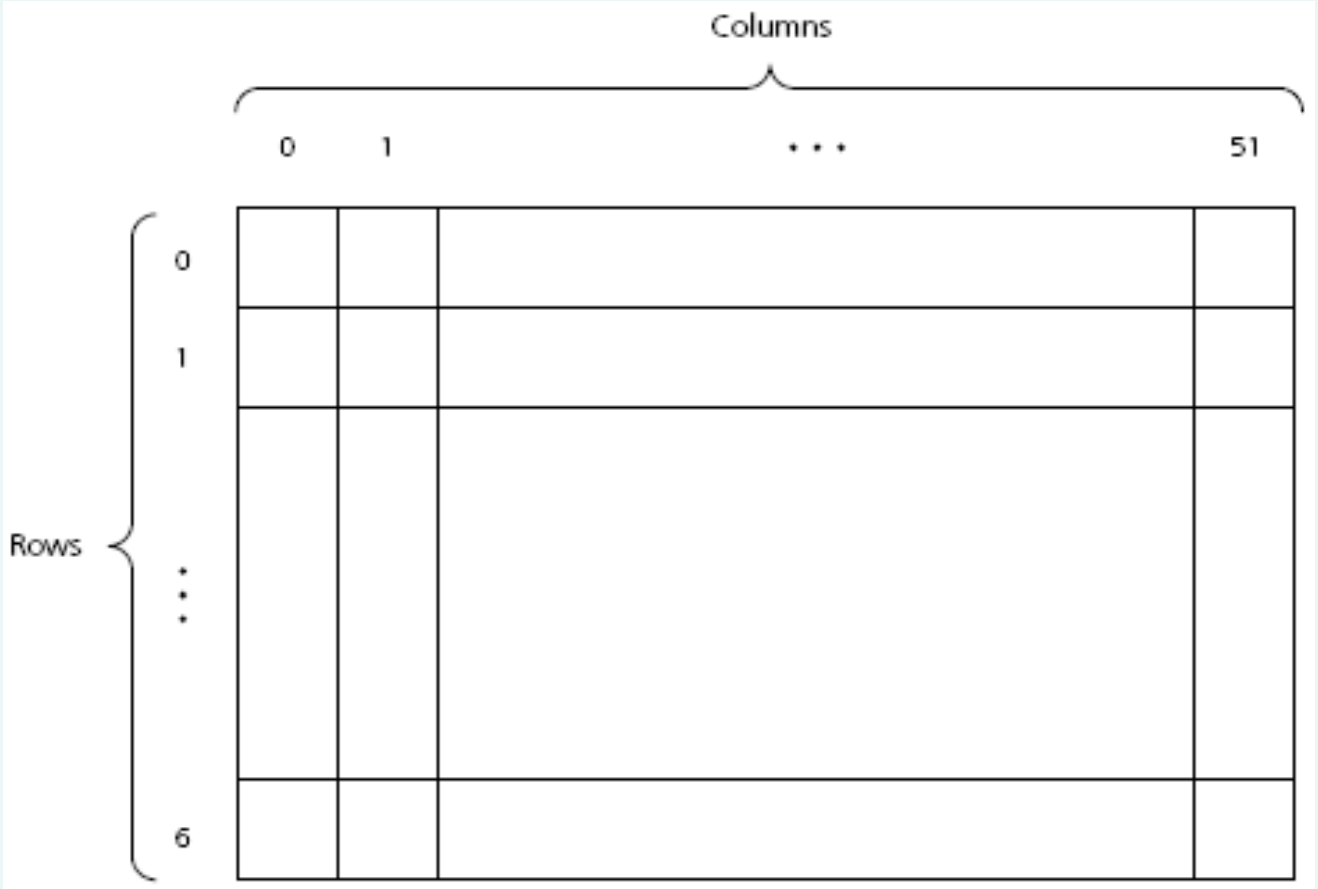


Figure 1-8

A two-dimensional array

Arrays

- Passing an array to a method
  - Declare the method as follows:

```
public double averageTemp(double[] temps, int n)
```

  - Invoke the method by writing:

```
double avg = averageTemp(maxTemps, 6);
```

  - Location of array is passed to the method
    - Cannot return a new array through this value
  - Method can modify content of the array

Selection Statements

- The `if` statement

```
if (expression)
   statement1
```

or

```
if (expression)
   statement1
else
   statement2
```

- Nested `if`

```
if (expression) {
   statement1
}
else if (expression) {
   statement2
}
else {
   statement3
} // end if
```

Selection Statements

- The `switch` statement

```
switch (integral expression) {
  case 1:
   statement1;
   break;
  case 2, case 3:
   statement2;
  case 4:
   statement3;
   break;
  default:
   statement4;
} //end of switch
```

Iteration Statements

- The `while` statement

```
while (expression) {
  statement
}
```

- `statement` is executed as long as `expression` is `true`
- `statement` may not be executed at all
- `continue` expression
  - Stops the current iteration of the loop and begins the next iteration at the top of the loop

Iteration Statements

- The `for` statement

```
for (initialize; test; update)
  statement
```

- `statement` is executed as long as `test` is `true`
- `for` statement is equivalent to a `while` statement
- The `for` loop and arrays

```
for (ArrayElementType variableName : arrayName)
  statement
```

Iteration Statements

- The `do` statement

```
do {
   statement
} while (expression);
```

- `statement` is executed until `expression` is `false`
- `do` statement loops at least once

Program Structure

- Typical Java program consists of
  - User written classes
  - Java Application Programming Interface (API) classes
- Java application
  - Has one class with a *main* method
- Java program basic elements:
  - Packages
  - Classes
  - Data fields
  - Methods

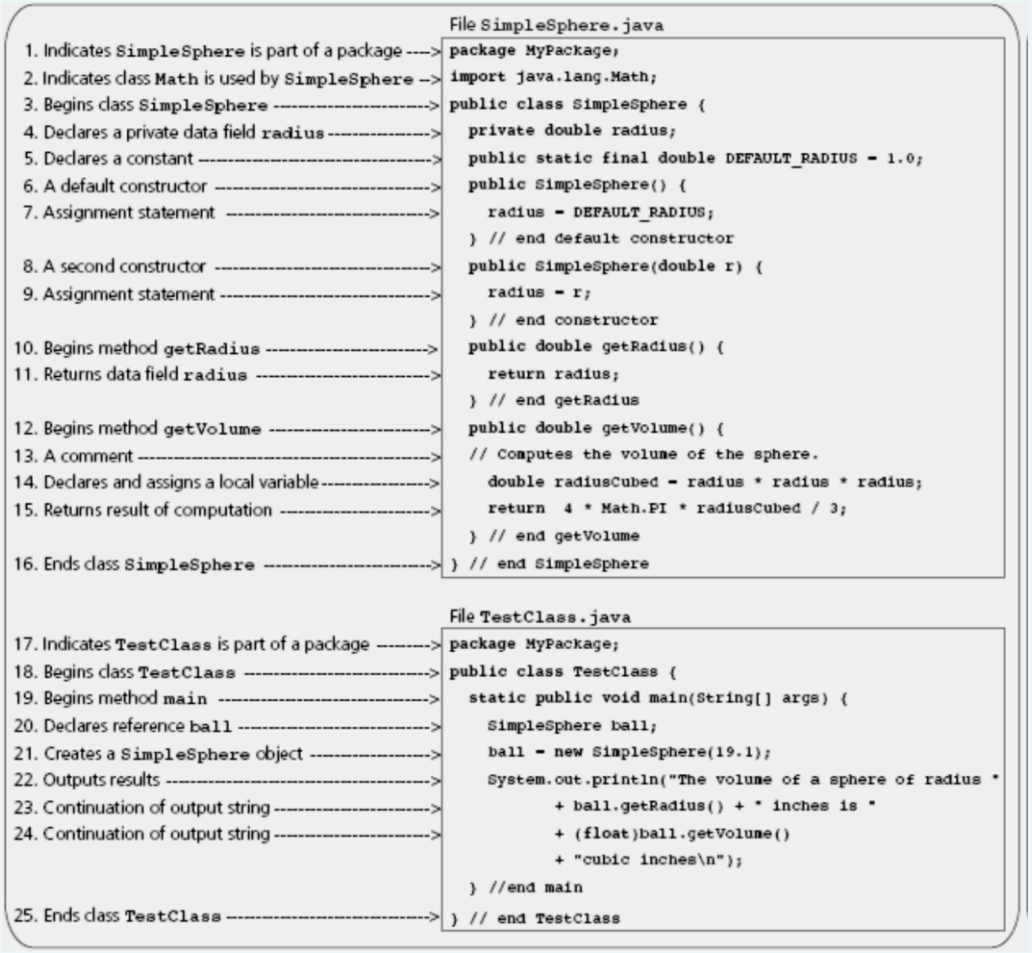
Packages

- Provide a mechanism for grouping related classes
- `package` statement
  - Indicates a class is part of a package
- Java assumes all classes in a particular package are contained in same directory
- Java API consists of many predefined packages

Packages

- `import` statement
  - Allows you to use classes contained in other packages
- Package `java.lang` is implicitly imported to all Java code

Packages



File SimpleSphere.java

```
1. Indicates SimpleSphere is part of a package ---->  package MyPackage;
2. Indicates class Math is used by SimpleSphere -->  import java.lang.Math;
3. Begins class SimpleSphere ---------------------->  public class SimpleSphere {
4. Declares a private data field radius ----------->    private double radius;
5. Declares a constant ---------------------------->    public static final double DEFAULT_RADIUS = 1.0;
6. A default constructor -------------------------->    public SimpleSphere() {
7. Assignment statement --------------------------->      radius = DEFAULT_RADIUS;
                                                        } // end default constructor
8. A second constructor --------------------------->    public SimpleSphere(double r) {
9. Assignment statement --------------------------->      radius = r;
                                                        } // end constructor
10. Begins method getRadius ----------------------->    public double getRadius() {
11. Returns data field radius --------------------->      return radius;
                                                        } // end getRadius
12. Begins method getVolume ----------------------->    public double getVolume() {
13. A comment ------------------------------------->    // Computes the volume of the sphere.
14. Declares and assigns a local variable --------->      double radiusCubed = radius * radius * radius;
15. Returns result of computation ----------------->      return  4 * Math.PI * radiusCubed / 3;
                                                        } // end getVolume
16. Ends class SimpleSphere ----------------------> } // end SimpleSphere
```

File TestClass.java

```
17. Indicates TestClass is part of a package -------->  package MyPackage;
18. Begins class TestClass ------------------------->  public class TestClass {
19. Begins method main ----------------------------->    static public void main(String[] args) {
20. Declares reference ball ------------------------>      SimpleSphere ball;
21. Creates a SimpleSphere object ----------------->      ball = new SimpleSphere(19.1);
22. Outputs results -------------------------------->      System.out.println("The volume of a sphere of radius "
23. Continuation of output string ------------------>              + ball.getRadius() + " inches is "
24. Continuation of output string ------------------>              + (float)ball.getVolume()
                                                                  + "cubic inches\n");
                                                          } //end main
25. Ends class TestClass --------------------------> } // end TestClass
```

Figure 1-1

A simple Java Program

Classes

- Data type that specifies data and methods available for instances of the class
- An object in Java is an instance of a class
- Class definition includes
  - Optional subclassing modifier
  - Optional access modifier
  - Keyword `class`
  - Optional `extends` clause
  - Optional `implements` clause
  - Class body

Classes

- Every Java class is a subclass of either
  - Another Java class
  - Object class
- `new` operator
  - Creates an object or instance of a class

Classes

Component	Syntax	Description
Subclassing modifier (use only one)	**`abstract`**	Class must be extended to be useful.
	`final`	Class cannot be extended.
Access modifiers	**`public`**	Class is available outside of package.
	no access modifier	Class is available only within package.
Keyword **`class`**	**`class`** *`class-name`*	Class should be contained in a file called *`class-name.java`*.
`extends` clause	**`extends`** *`superclass-name`*	Indicates that this class is a subclass of the class *`superclass-name`* in the *`extends`* clause.
`implements` clause	**`implements`** *`interface-list`*	Indicates the interfaces that this class implements. The *`interface-list`* is a comma-separated list of interface names.
Class body	Enclosed in braces	Contains data fields and methods for the class.

Figure 1-2

Components of a class

Data Fields

- Class members that are either variables or constants
- Data field declarations can contain
  - Access modifiers
  - Use modifiers
  - Modules

Data Fields

Type of modifier	Keyword	Description
Access modifier (use only one)	`public`	Data field is available everywhere (when the class is also declared `public`).
	`private`	Data field is available only within the class.
	`protected`	Data field is available within the class, available in subclasses, and available to classes within the same package.
	No access modifier	Data field is available within the class and within the package.
Use modifiers (all can be used at once)	`static`	Indicates that only one such data field is available for all instances of this class. Without this modifier, each instance has its own copy of a data field.
	`final`	The value provided for the data field cannot be modified (a constant).
	`transient`	The data field is not part of the persistent state of the object.
	`volatile`	The value provided for the data field can be accessed by multiple threads of control. Java ensures that the freshest copy of the data field is always used.

Figure 1-3

Modifiers used in data field declarations

Methods

- Used to implement operations
- Should perform one well-defined task
- Method modifiers
  - Access modifiers and use modifiers
- Valued method
  - Returns a value
  - Body must contain **`return`** `expression;`

Method Modifiers

Type of modifier	Keyword	Description
Access modifier (use only one)	`public`	Method is available everywhere (when the class is also declared as `public`).
	`private`	Method is available only within the class (cannot be declared `abstract`).
	`protected`	Method is available within the class, available in subclasses, and available to classes within the same package.
	No access modifier	Method is available within the class and to classes within the package.
Use modifiers (all can be used at once)	`static`	Indicates that only one such method is available for all instances of this class. Since a `static` method is shared by all instances, the method can refer only to data fields that are also declared `static` and shared by all instances.
	`final`	The method cannot be overridden in a subclass.
	`abstract`	The method must be overridden in a subclass.
	`native`	The body of the method is not written in Java but in some other programming language.
	`synchronized`	The method can be run by only one thread of control at a time.

Figure 1-4

Modifiers used in a method declaration

Methods

- Syntax of a method declaration

```
access-modifier use-modifiers return-type
          method-name (formal-parameter-list) {
  method-body
}
```

- Arguments are passed by value
  - Except for objects and arrays
    - A reference value is copied instead
- Java 1.5 allows a method to have a variable number of arguments of the same type
  - Using the ellipses (three consecutive dots)

Methods

- Constructor
  - Special kind of method
  - Has the same name as the class and no return type
  - Executed only when an object is created
- A class can contain multiple constructors

How to Access Members of an Object

- Data fields and methods declared *public*
  - Name the object, followed by a period, followed by member name
- Members declared *static*
  - Use the class name, followed by a period, followed by member name

Inheritance

- Technique for creating a new class that is based on one that already exists.
  - Desire to add new features
  - Desire to define a more specific data type
  - We don't want to change the original class
- Example: SimpleSphere and ColoredSphere
  - We already have the SimpleSphere class
  - ColoredSphere will be everything a SimpleSphere is, but more.

Inheritance

- Terminology
  - Base class (or superclass): the original class from which we create the new one
  - Derived class (or subclass): the new class we create
  - We say that the subclass inherits data members and operations of its superclass.
- Accessibility
  - Subclass has access to attributes of its superclass, but the superclass cannot access attributes of its subclass(s)

Inheritance

- How to define

```
public class ColoredSphere extends SimpleSphere
```

  - The Java keyword `extends` means we are using inheritance.
- Constructor for the derived class:

```
public ColoredSphere(Color c) {
  super();     // We call superclass constructor
  color = c;
}
```

Inheritance

- Another use of the word `super`
  - If we write code inside ColoredSphere that requires us to call a method in the superclass SimpleSphere, such as getVolume.

```
double myVolume = super.getVolume();
```

- If a client class uses a ColoredSphere object, it can use a superclass method automatically.

```
double volume = cs.getVolume();
```

  - This is a legal statement even though getVolume is not inside ColoredSphere.java: it's inherited.

Useful Java Classes

- The `Object` class
  - Java supports a single class inheritance hierarchy
    - With class `Object` as the root
  - More useful methods
    - **`public boolean`** `equals(Object obj)`
    - **`protected void`** `finalize()`
    - **`public int`** `hashCode()`
    - **`public String`** `toString()`

Useful Java Classes

- The `Arrays` class
  - `import java.util.Arrays;`
  - Contains static methods for manipulating arrays
- Commonly used examples
  - Sort (does it in ascending order)
  - Binary search (quickly finds a value in the array)
  - toString
- Example: Let's say `a` is an array of 1000 `int`s

```
Arrays.sort(a);
```

Useful Java Classes

- String classes
  - Class `String`
    - Declaration examples:
      - `String title;`
      - `String title = "Walls and Mirrors";`
    - Assignment example:
      - `Title = "Walls and Mirrors";`
    - String length example:
      - `title.length();`
    - Referencing a single character
      - `title.charAt(0);`
    - Comparing strings
      - `title.compareTo(string2);`

Useful Java Classes

- String classes (continued)
  - Class `String`
    - Concatenation example:

```
String monthName = "December";
int day = 31;
int year = 02;
String date = monthName + " " + day + ", 20"
  + year;
```

Useful Java Classes

- String classes (continued)
  - Class `StringBuffer`
    - Creates mutable strings
    - Provides same functionality as class `String`
    - More useful methods
      - `public StringBuffer append(String str)`
      - `public StringBuffer insert(int offset, String str)`
      - `public StringBuffer delete(int start, int end)`
      - `public void setCharAt(int index, char ch)`
      - `public StringBuffer replace(int start, int end, String str)`

Useful Java Classes

- String classes (continued)
  - Class `StringTokenizer`
    - Allows a program to break a string into pieces or tokens
    - More useful methods
      - `public StringTokenizer(String str)`
      - `public StringTokenizer(String str, String delim)`
      - `public StringTokenizer(String str, String delim, boolean returnTokens)`
      - `public String nextToken()`
      - `public boolean hasMoreTokens()`

Java Exceptions

- Exception
  - Handles an error during execution
- Throw an exception
  - To indicate an error during a method execution
- Catch an exception
  - To deal with the error condition

Catching Exceptions

- Java provides `try-catch` blocks
  - To handle an exception
- Place statement that might throw an exception within the `try` block
  - Must be followed by one or more `catch` blocks
  - When an exception occurs, control is passed to `catch` block
- `Catch` block indicates type of exception you want to handle

Catching Exceptions

- `try-catch` blocks syntax

```
try {
  statement(s);
}
catch (exceptionClass identifier) {
  statement(s);
}
```

- Some exceptions from the Java API cannot be totally ignored
  - You must provide a handler for that exception

Catching Exceptions

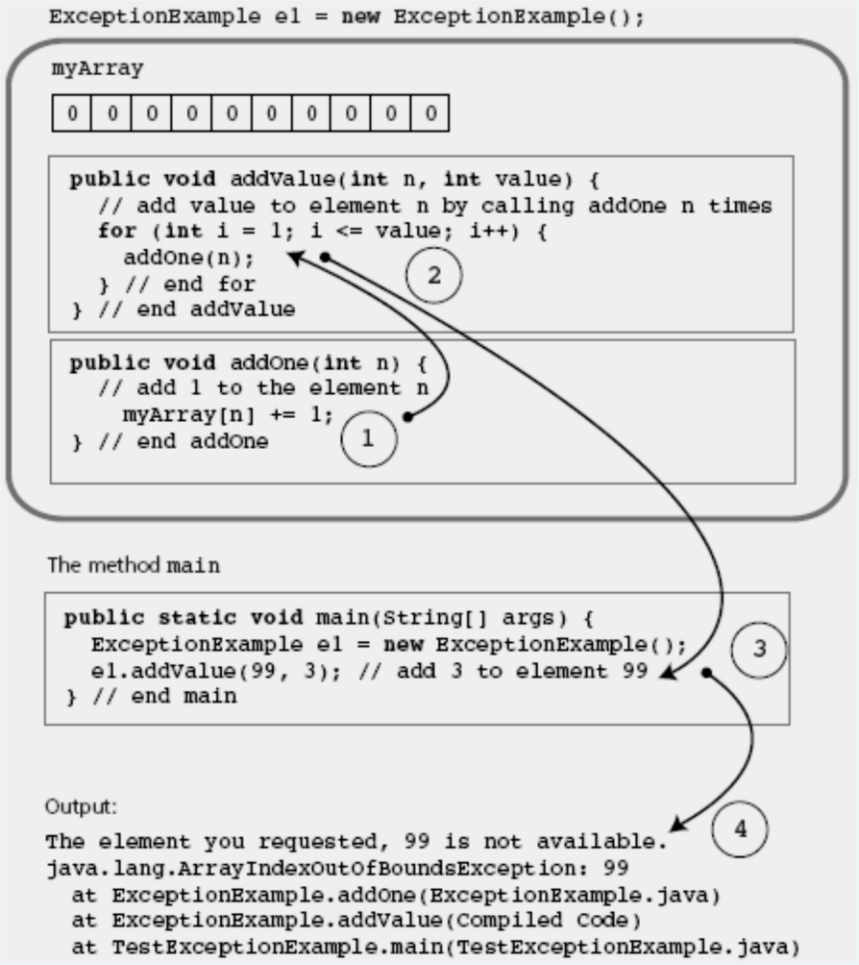


Figure 1-9

Flow of control in a simple Java application

Catching Exceptions

- Types of exception
  - Checked exceptions
    - Instances of classes that are subclasses of `java.lang.Exception`
    - Must be handled locally or thrown by the method
    - Used when method encounters a serious problem
  - Runtime exceptions
    - Occur when the error is not considered serious
    - Instances of classes that are subclasses of `java.lang.RuntimeException`

Catching Exceptions

- The `finally` block
  - Executed whether or not an exception is thrown
  - Can be used even if no `catch` block is used
  - Syntax

```
finally {
  statement(s);
}
```

Throwing Exceptions

- `throws` clause
  - Indicates a method may throw an exception
    - If an error occurs during its execution
  - Syntax

```
public methodName throws ExceptionClassName
```

- `throw` statement
  - Used to throw an exception at any time
  - Syntax

```
throw new exceptionClass(stringArgument);
```

- You can define your own exception class

Text Input and Output

- Input and output consist of streams
- Streams
  - Sequence of characters that either come from or go to an I/O device
  - `InputStream` - Input stream class
  - `PrintStream` - Output stream class
- java.lang.System provides three stream variables
  - `System.in` – standard input stream
  - `System.out` – standard output stream
  - `System.err` – standard error stream

Input

- Character streams

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

String nextLine = stdin.readLine();

StringTokenizer input = new
  StringTokenizer(nextLine);
x = Integer.parseInt(input.nextToken());
y = Integer.parseInt(input.nextToken());
```

Input

- Another approach: the `Scanner` class

```
int nextValue;
int sum=0;
Scanner kbInput = new Scanner(System.in);
nextValue = kbInput.nextInt();
while (nextValue > 0) {
  sum += nextValue;
  nextValue = kbInput.nextInt();
} // end while
kbInput.close();
```

Input

- The `Scanner` class (continued)
  - More useful `next` methods
    - `String next();`
    - `boolean nextBoolean();`
    - `double nextDouble();`
    - `float nextFloat();`
    - `int nextInt();`
    - `String nextLine();`
    - `long nextLong();`
    - `short nextShort();`

Output

- Methods `print` and `println`
  - Write character strings, primitive types, and objects to `System.out`
  - `println` terminates a line of output so next one starts on the next line
  - When an object is used with these methods
    - Return value of object's `toString` method is displayed
    - You usually override this method with your own implementation
  - Problem
    - Lack of formatting abilities

Output

- Method `printf`
  - C-style formatted output method
  - Syntax

```
printf(String format, Object... args)
```

  - Example:

```
String name = "Jamie";
int x = 5, y = 6;
int sum = x + y;
System.out.printf("%s, %d + %d = %d", name,
  x, y, sum);
//produces output Jamie, 5 + 6 = 11
```

Output

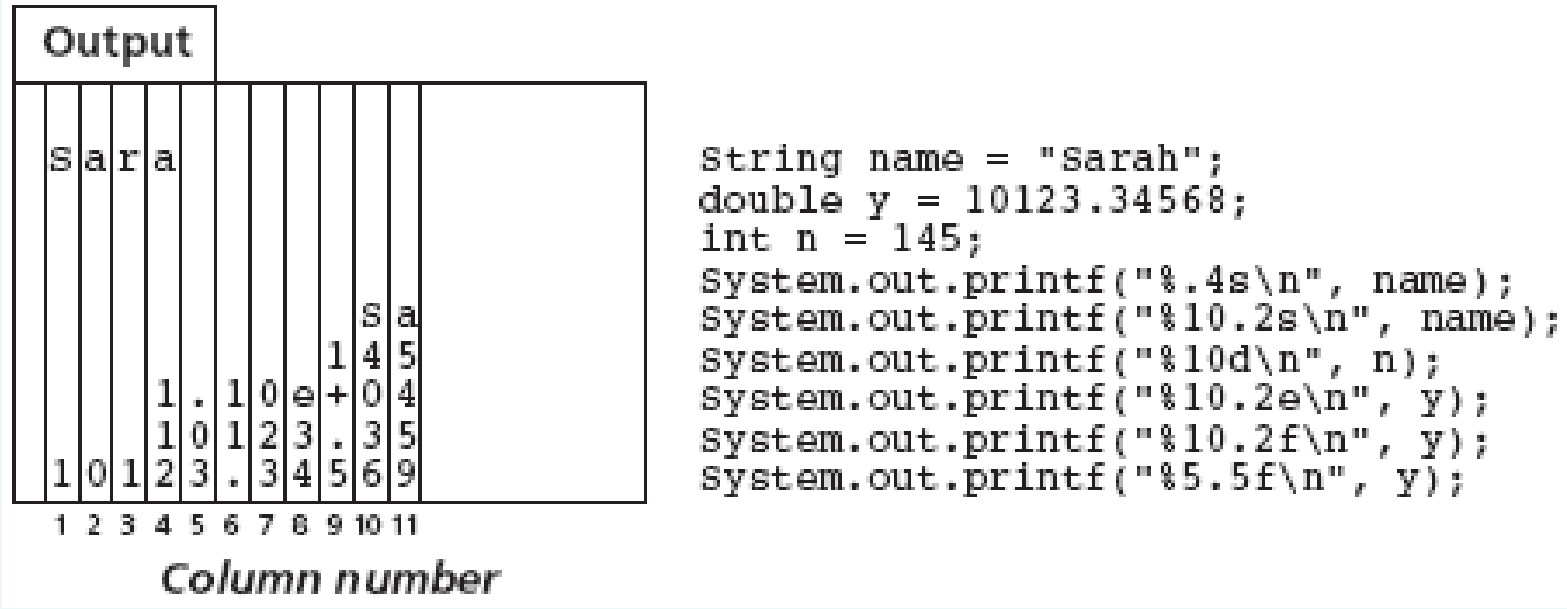


Figure 1-10

Formatting example with `printf`

The Console Class

- `import java.io.Console;`
- Initialize: `Console cons = System.console();`
  - Returns null if no console available (e.g. in IDE instead of command line)
- Can format output using `printf()`
- Input
  - Has `readLine()` method that can read formatted input, in an analogous manner to `printf()` for output.
  - `readPassword()`: read input without echoing what the user types in.

File Input and Output

- File
  - Sequence of components of the same type that resides in auxiliary storage
  - Can be large and exists after program execution terminates
- Files vs. arrays
  - Files grow in size as needed; arrays have a fixed size
  - Files provides both sequential and random access; arrays provide random access
- File types
  - Text and binary (general or nontext) files

Text Files

- Designed for easy communication with people
  - Flexible and easy to use
  - Not efficient with respect to computer time and storage
- End-of-line symbol
  - Creates the illusion that a text file contains lines
- End-of-file symbol
  - Follows the last component in a file
- Scanner class can be used to process text files

Text Files

Figure 1-11

A text file with end-of-line and end-of-file symbols

Text Files

- Example

```
String fname, lname;
int age;
Scanner fileInput;
File inFile = new File("Ages.dat");
try {
   fileInput = new Scanner(inFile);
   while (fileInput.hasNext()) {
     fname = fileInput.next();
     lname = fileInput.next();
     age = fileInput.nextInt();
     age = fileInput.nextInt();
     System.out.printf("%s %s is %d years old.\n",
     fname, lname, age);
   } // end while
   fileInput.close();
} // end try
catch (FileNotFoundException e) {
   System.out.println(e);
} // end catch
```

Text Files

- Open a stream to a file
  - Before you can read from or write to a file
  - Use class `FileReader`
    - Constructor throws a `FileNotFoundException`
  - Stream is usually embedded within an instance of class `BufferedReader`
    - That provides text processing capabilities
  - `StringTokenizer`
    - Used to break up the string returned by `readLine` into tokens for easier processing

Text Files

- Example

```
BufferedReader input;
StringTokenizer line;
String inputLine;
try {
   input = new BufferedReader(new FileReader("Ages.dat"));
   while ((inputLine = input.readLine()) != null) {
    line = new StringTokenizer(inputLine);
    // process line of data
    ...
   }
} // end try
catch (IOException e) {
   System.out.println(e);
   System.exit(1); // I/O error, exit the program
} // end catch
```

Text Files

- File output
  - You need to open an output stream to the file
  - Use class `FileWriter`
  - Stream is usually embedded within an instance of class `PrintWriter`
    - That provides methods `print` and `println`

Text Files

- Example

```
try {
   PrintWriter output = new PrintWriter(new
    FileWriter("Results.dat"));
   output.println("Results of the survey");
   output.println("Number of males: " + numMales);
   output.println("Number of females: " +
   numFemales);
   // other code and output appears here...
} // end try
catch (IOException e) {
   System.out.println(e);
   System.exit(1); // I/O error, exit the program
} // end catch
```

Text Files

- Closing a file
  - Syntax

```
myStream.close();
```

- Adding to a text file
  - When opening a file, you can specify if file should be replaced or appended
  - Syntax

```
PrintWriter ofStream = new PrintWriter(new
  FileOutputStream("Results.dat", true));
```

Object Serialization

- Data persistence
  - Data stored in a file for later use
- Object serialization
  - Java mechanism to create persistent objects
- Serialization
  - Transforming an object into a sequence of bytes that represents the object
  - Serialized objects can be stored to files for later use

Object Serialization

- Deserialization
  - Reverse process
- Interface `java.io.Serializable`
  - Needed to save an object using object serialization
  - Contains no methods
- Objects referenced by a serialized object are also serialized
  - As long as these objects also implement the `Serializable` interface

Summary

- Java packages
  - Provide a mechanism for grouping related classes
- `import` statement
  - Required to use classes contained in other packages
- Object in Java is an instance of a class
- Class
  - Data type that specifies data and methods available
  - Data fields are either variables or constants
  - Methods implement object behavior
- Method parameters are passed by value

Summary

- Comments in Java
  - Comment lines
  - Multiple-line comments
- Java identifier
  - Sequence of letters, digits, underscores, and dollar signs
- Primitive data types categories
  - Integer, character, floating point, and boolean
- Java reference
  - Used to locate an object

Summary

- Define named constant with `final` keyword
- Java uses short-circuit evaluation for logical and relational expressions
- Array
  - Collection of references that have the same data type
- Selection statements
  - `if` and `switch`
- Iteration statements
  - `while`, `for`, and `do`

Summary

- String
  - Sequence of characters
  - String classes: `String`, `StringBuffer`, `StringTokenizer`
- Exceptions
  - Used to handle errors during execution
- Files are accessed using `Scanner` class or streams
- Data persistence and object serialization