Introduction to Java Programming

What is Java?

Getting Started

Objectives

At the end of this module you should be able to

- Understand Java
- Discuss why Java should be used and who owns it
- Talk about Java Classifications
- Use SDK & JRE
- Create Jar files
- Use the Java Programming Model
- Create Java Applications
- Create Java Applets
- Compile and run Java Applications
- Compile and run Java Applets

What is Java?

- An Object Oriented Programming Language
- Introduced in 1995 by Sun Microsystems
- Platform Independent
- Syntactically very similar to C, C++, C#
- A collection of APIs

Why Use JAVA?

- An Object Oriented Programming Language
  - Manage complexity
  - Enable code reuse
  - Create flexibility
  - Etc.
- Platform Independent
  - Sun says: Write once run anywhere
  - Reality: Write once test everywhere
- Comes with large collection of APIs
- Built-in features like:
  - Security
  - Networking
  - Memory Management including garbage collection
- Open Source and Free (Sun Community Source License)
- Extremely popular for web and enterprise based applications

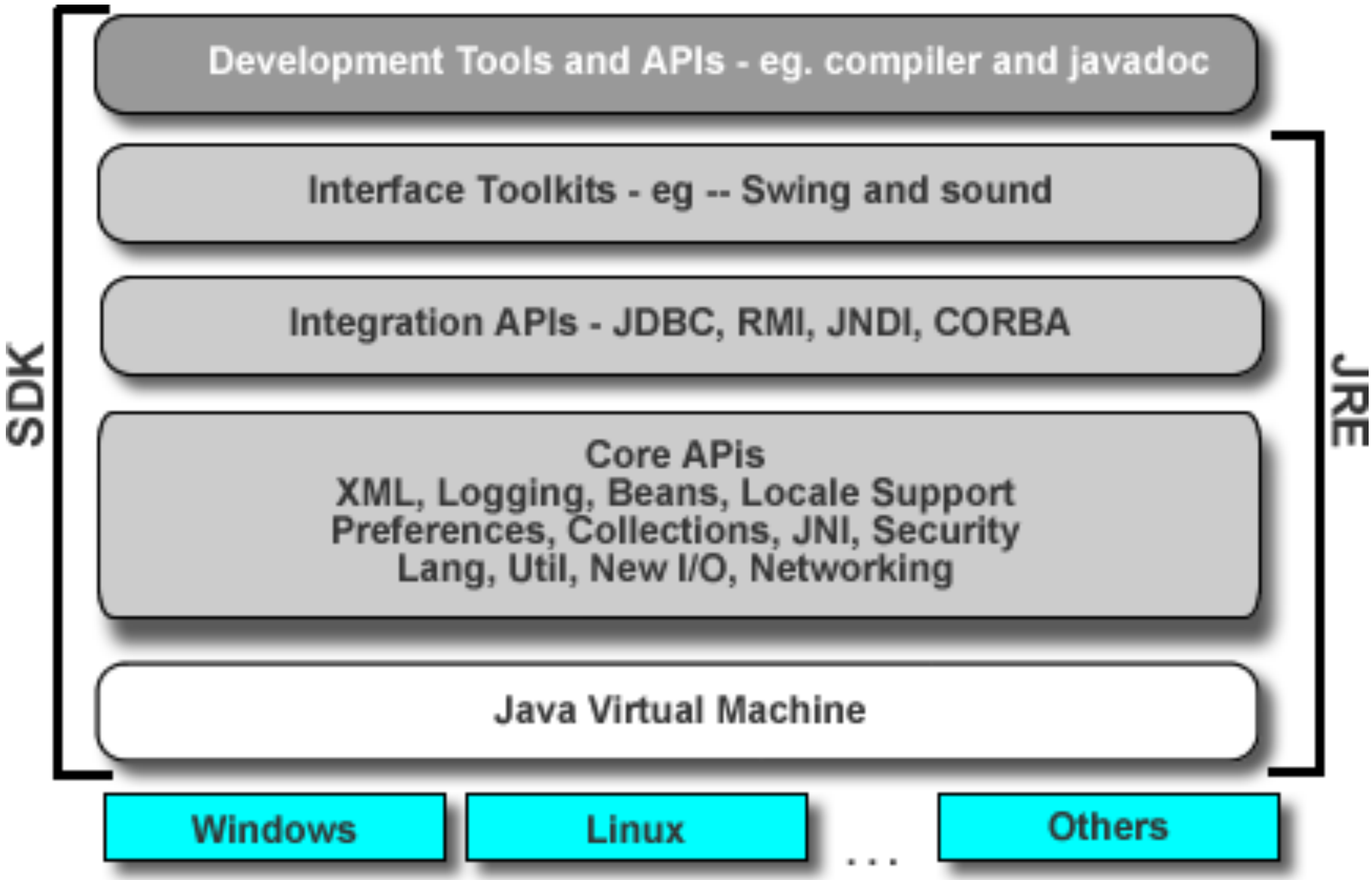
Who Owns Java?

- Nobody and everybody
- Sun Microsystems owns the right to the Java name
- Sun provides standard implementations i.e. Java SE
- Java Community Process (JCP)

Java Classifications

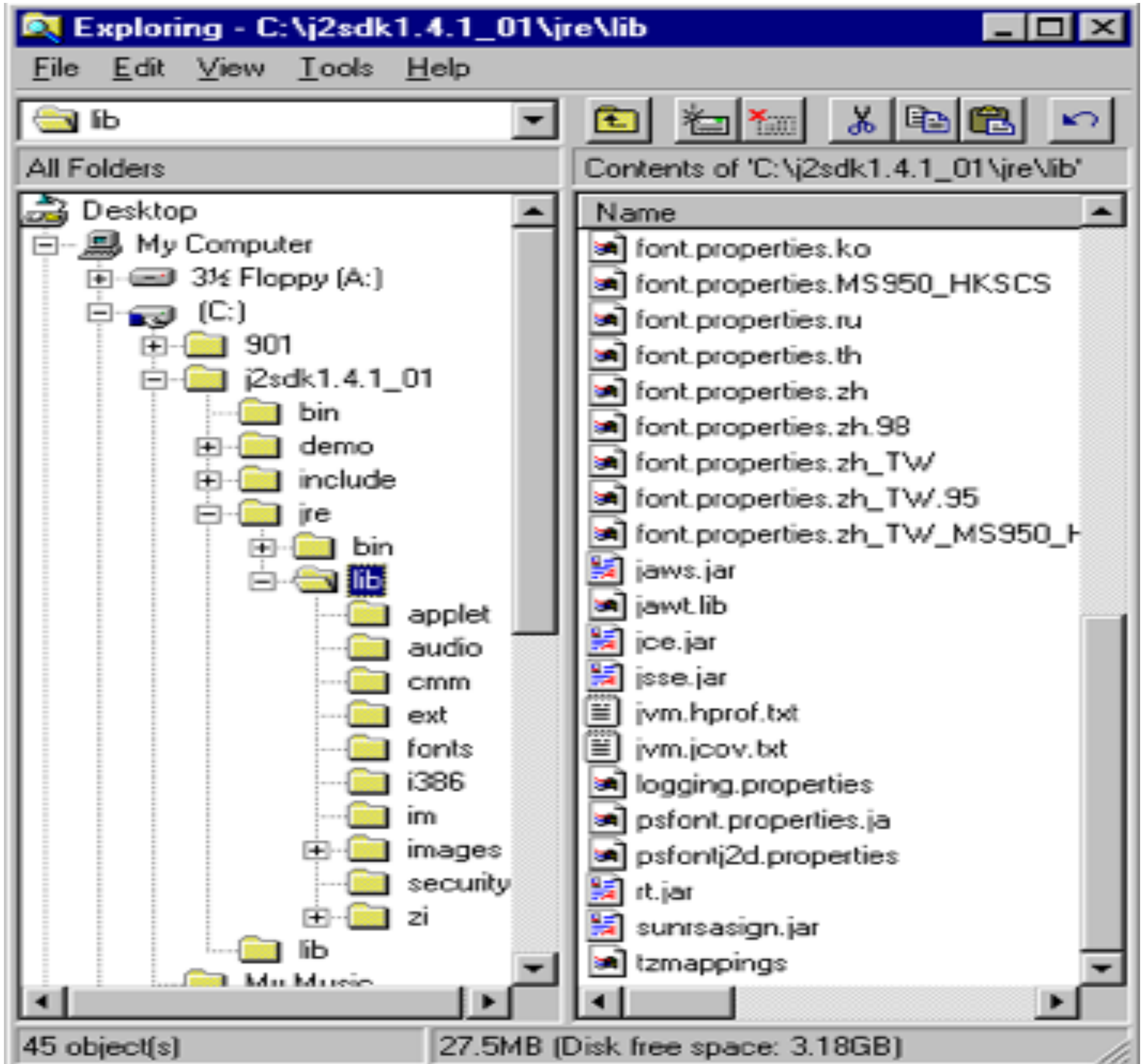
- Java SE
  - Targeted at personal, workstation, and server use
  - Considered the “core”
  - Provides the standard execution platform
  - Current version is 1.7
- Java Enterprise Edition (J2EE)
  - Targeted at server use
  - Enterprise and web extensions to the “core”
  - Utilizes Java SE as the execution platform
  - Current version is 1.6
- Java Micro Edition (J2ME)
  - Targeted at embedded devices
  - Utilizes a subset of the Java SE as execution platform

SDK & JRE



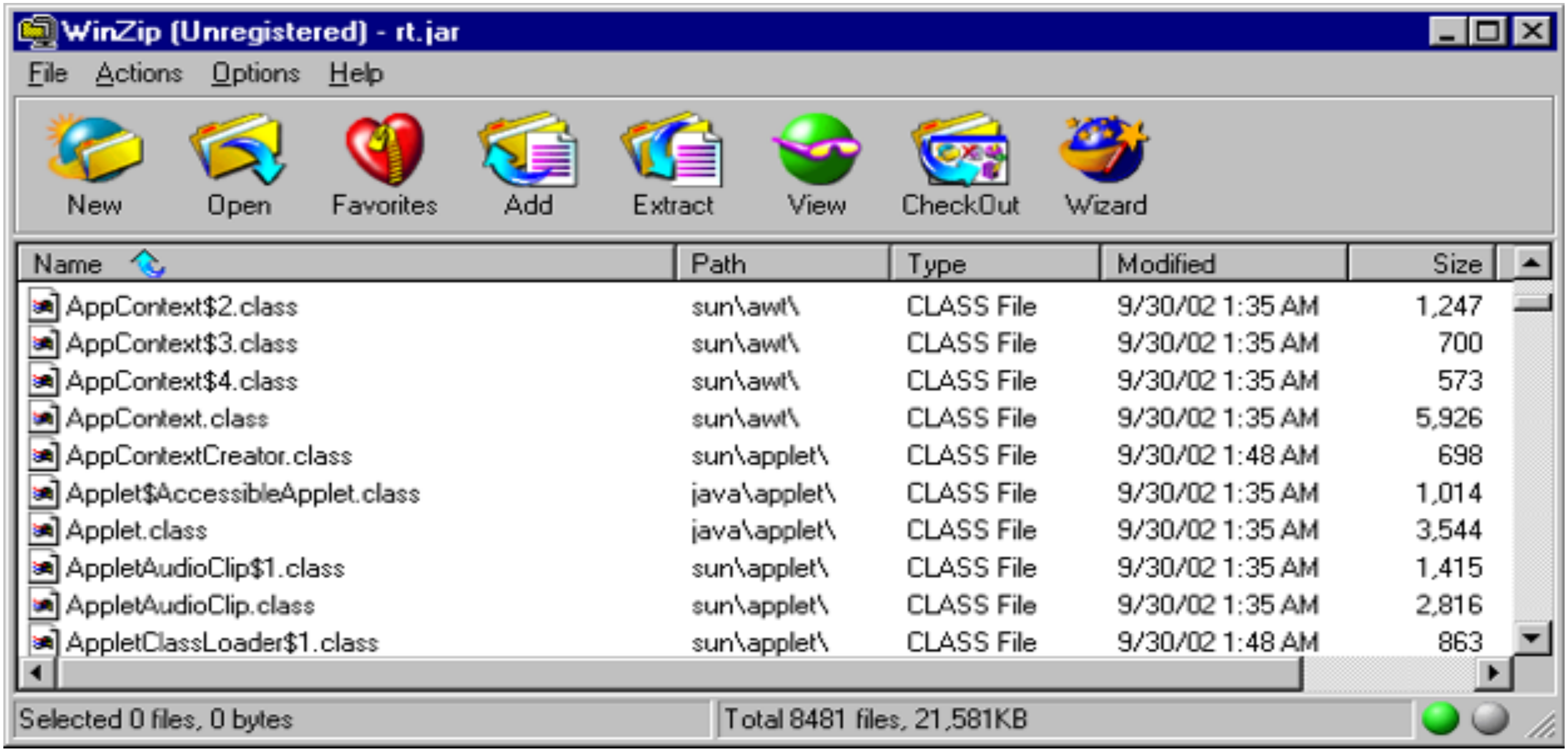
SDK
Development Tools and APIs - eg. compiler and javadoc
Interface Toolkits - eg -- Swing and sound
Integration APIs - JDBC, RMI, JNDI, CORBA
Core APIs
XML, Logging, Beans, Locale Support
Preferences, Collections, JNI, Security
Lang, Util, New I/O, Networking
Java Virtual Machine
JRE
Windows
Linux
...
Others

SDK & JRE (continued)



rt.jar (formally classes.zip)

The execution environment libraries are bundled in `rt.jar`

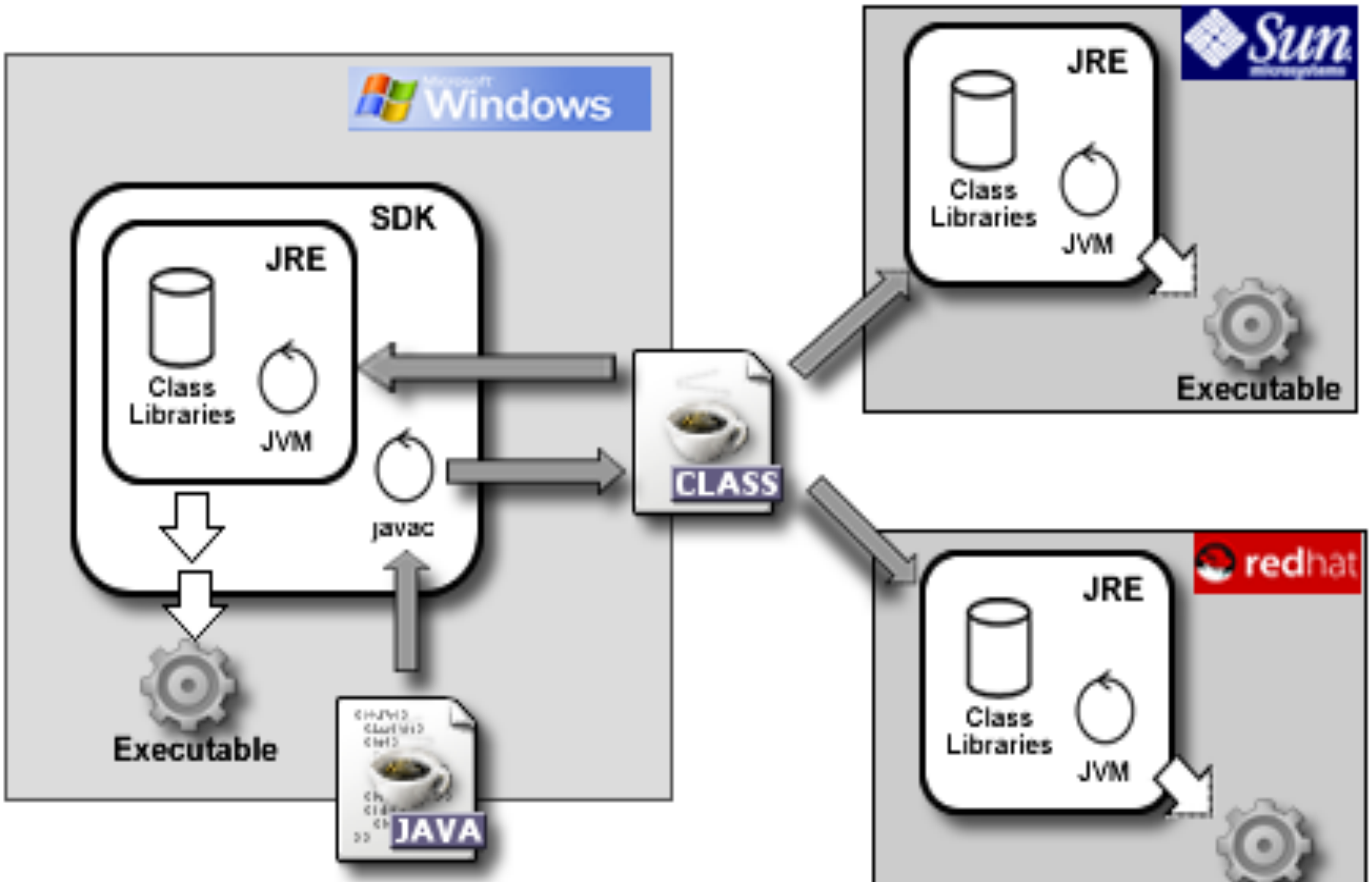


Java Programming Model

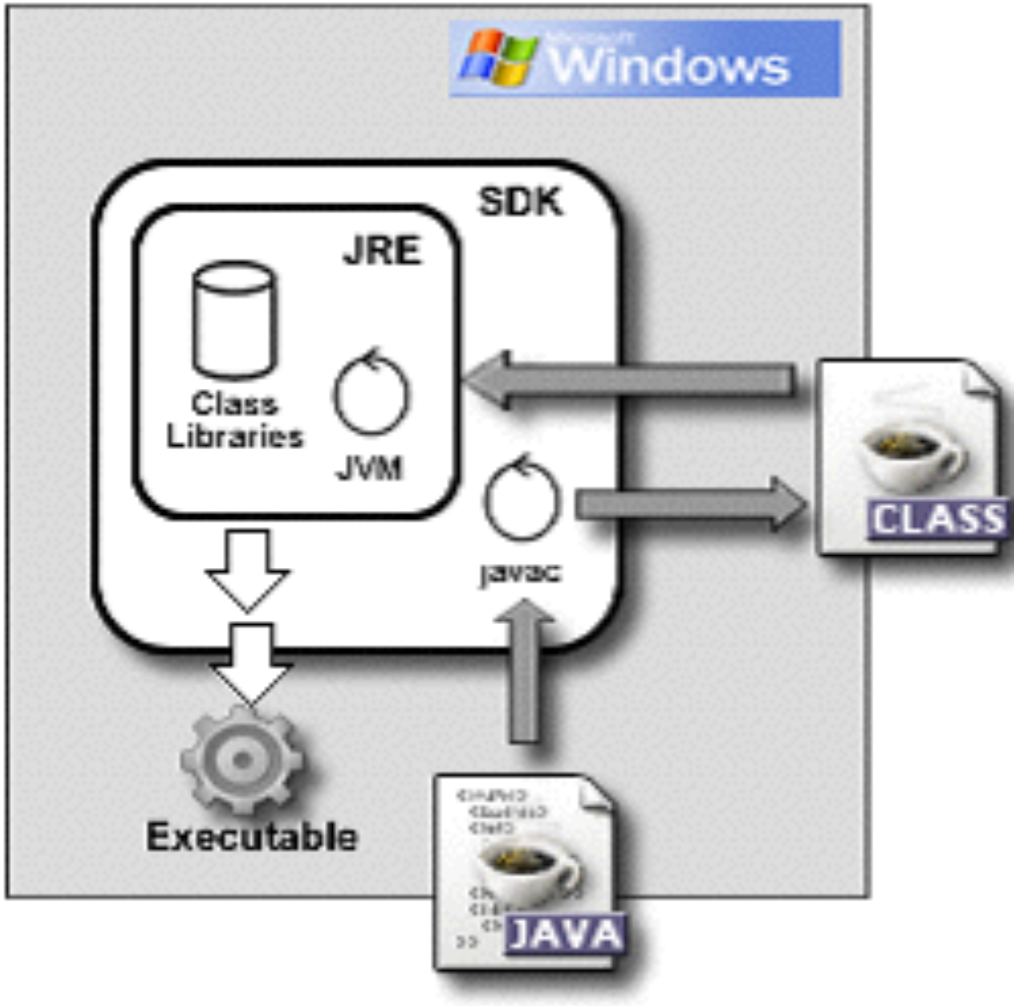
The basic programming model steps:

1. Create source code
   - Stored as test file
   - Has extension .java
2. Compile source code
   - Utilize java compiler - javac
   - Does syntax and language validation
   - Generates platform independent bytecode
   - Stored in a .class file
3. Distribute .class files
   - On the web (for applets)
   - On the server (for enterprise applications)
   - On the client (for applications)
4. Execute the “application”
   - Use the Java Runtime Environment (JRE)
   - JRE utilizes a Java Virtual Machine (JVM)
   - JVM is responsible for loading application and executing it

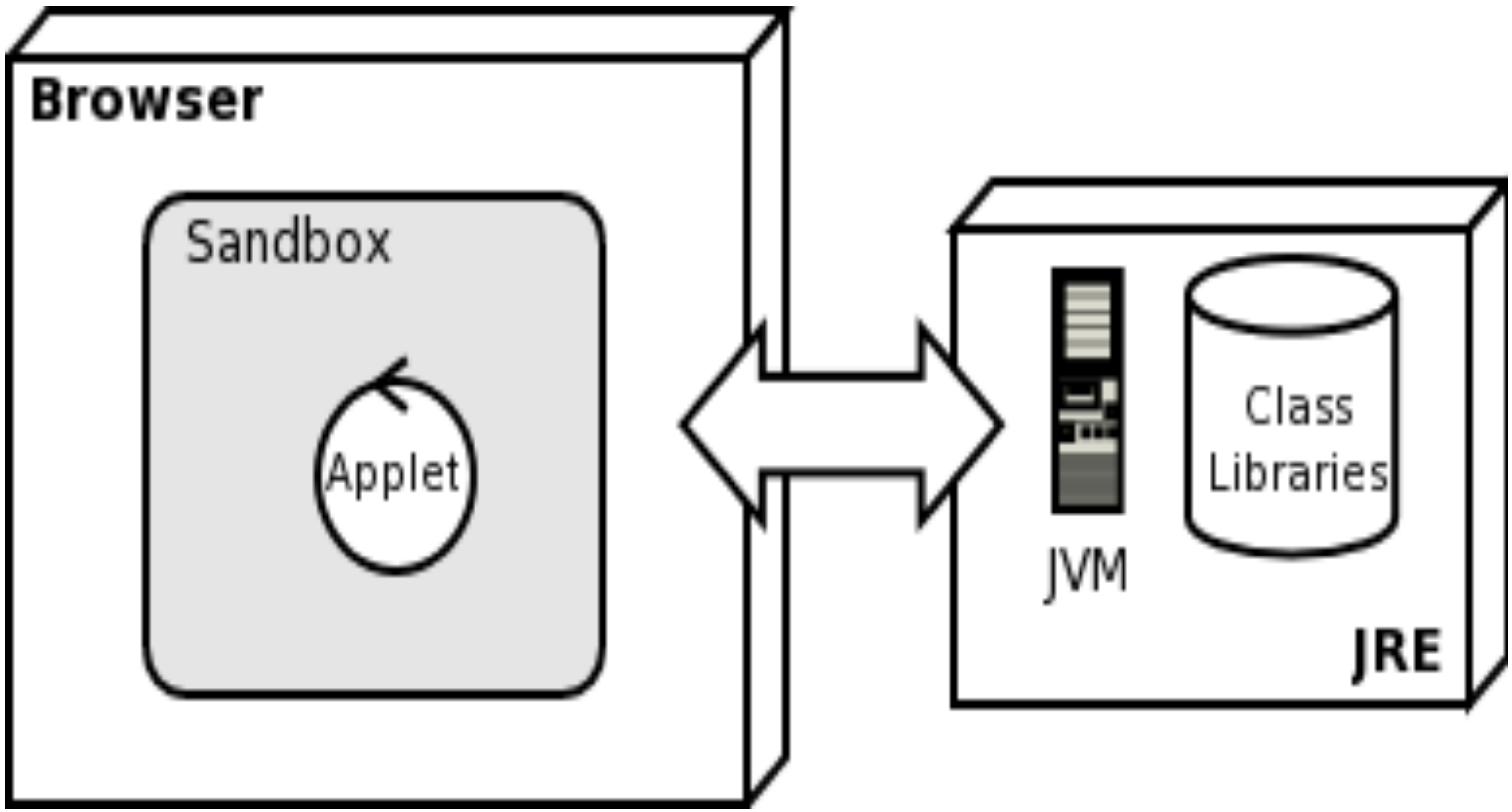
Java Programming Model



Stand-alone Java Application



Web-based Java Application - Applet



Compiling and Executing Java Applications

Developing

HelloWorld.java

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

```
> javac HelloWorld.java
```

```
HelloWorld.class
```

Executing

```
> java HelloWorld
```

```
Hello World!
```

Compiling and Executing Java Applets

Developing

HelloWorldApplet.java

```
import java.applet.*;
import java.awt.*;

public class HelloWorldApplet extends Applet {
  public void paint(Graphics g) {
    g.drawString("Hello World!",50,50);
  }
}
```

```
>javac HelloWorldApplet.java
```

```
HelloWorldApplet.class
```

HelloWorld.html

```
<html>
<head><title> Hello World Applet </title></head>
<body>
<applet code="HelloWorldApplet.class"
        width="150"
        height = "50">
</applet>
</body>
<html>
```

Executing

```
> appletviewer HelloWorld.html
```

Summary

We covered

- Understanding Java
- Why Java should be used and who owns it
- Java Classifications
- Using SDK & JRE
- Creating Jar files
- Using the Java Programming Model
- Creating Java Applications
- Creating Java Applets
- Compiling and running Java Applications
- Compiling and running Java Applets

OO Programming with Java

Objectives

At the end of this section you should be able to

- Discuss how a Java class definition is organized
- Write and compile Java class definitions
- Discuss what `init` and `main` methods do in Java
- Understand the purpose of an application class and a run-time container
- Use `javadoc` to create documentation

About Class Definitions

- The class definition starts with the class name
- Can be preceded by the optional keyword public
- We can name our classes whatever we want
  - Subject to a small set of rules about names
  - Class name cannot be a keyword
  - A keyword is a word reserved by Java, like `public` or `class`
- Normally we try to pick a meaningful name in the context of the application
- Typically follow Java conventions
  - Not required by Java, but good style
  - Capitalize the first letter of each word in the class name
  - e.g. `BankAccount`, `FullServiceATM`

About Class Definitions (cont.)

- The class definition contains a list of
  - Variable definitions
  - Method definitions
- The variable definitions and the method definitions can be placed in any order in the class definition
  - Order is not important like in structure programs
  - Java has no rules about order of definitions
- Java convention is to capitalize the first letter of each word in
  - A variable
    `accountNumber, ssn, firstName`
  - A method name except for the first one
    `getAccountNumber(), setFirstName(String s)`

About Class Definitions (cont.)

- The whole class definition is
  - Defined in a block of code “labeled” class
  - Whose body is contained in braces `{ }`
- Each method has a body
  - Enclosed in braces `{ }`
  - Called the method definition
  - Provides the execution logic for the method
- Statements end with a semi-colon
  - Variable declarations
  - Variable assignment
- Java is not white space sensitive, except when dealing with `String`

Class Definition Example

```
class A {
  private int i;
  public void setI(int j) {
    i = j;
  }
}
```

```
class B
{
  private int i;
  public void setI(int j)
  {
    i = j;
  }
}
```

Comments in Java

- Java, like all programming languages, uses comments to document source code
- Java uses the same two types of comments that appear in C++
  - `// single line comment`
  - ```
    /*  multi
        line
        comments */
    ```
- Also adds a third kind - the `javadoc` comment - which is unique to Java
  - ```
    /** multi
        line
        javadoc
        comment */
    ```

Comment Example

Example 2-2: Comments in Java Code

```
/**
* This is the start of a Javadoc comment
* This is the second line of a Javadoc comment
*/

class HelloWorld { // This is a single line comment
     //Another single line comment
     public static void main (String [] args) {
     /*Here is where we have used a multi-line
      comment to temporarily comment-out a piece of
      code
      System.out.println("I'm commented out"); */
    System.out.println("I'm not commented out");
  }
}
```

Using javadoc

- `javadoc` is one of the utilities that is provide in the SDK
  - Designed so that a programmer can document classes, methods, fields and packages in an application
  - Formats, organizes and cross-references the documentation
  - Generates HTML documentation
- There is a syntax that `javadoc` comments have to follow
  - Allows for complex formatting and cross-referencing
  - For example, the whole set of API documentation for the Java SE and the other Java 2 editions are generated using `javadoc`
  - There are built-in “tags” for things like
    - Version
    - Author
    - Date
    - Etc.

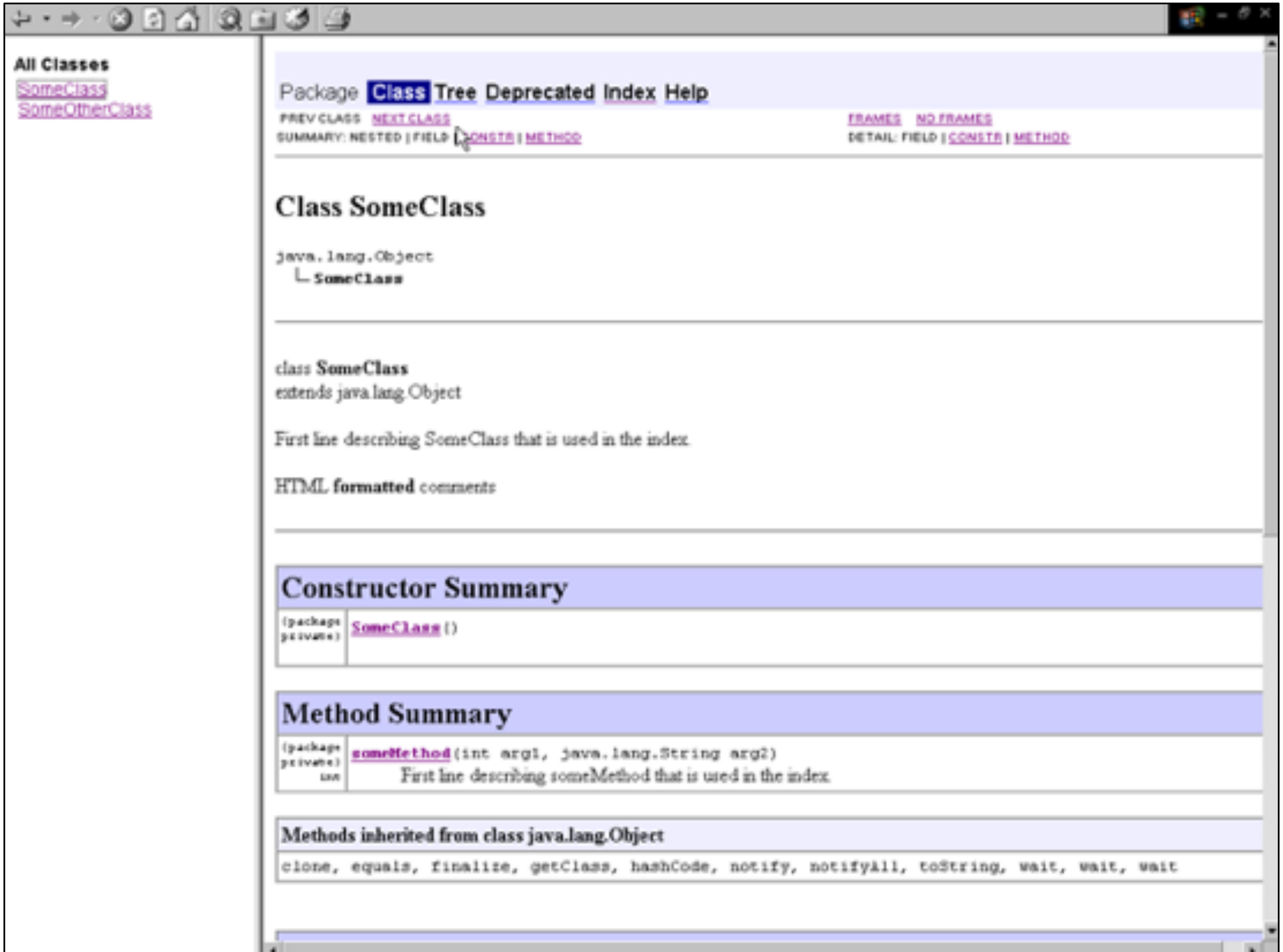
javadoc Example

Example 2-3: Javadoc documentation

```
/**
* First line describing SomeClass that is used in the index.
* <p>
* HTML <strong>formatted</strong> comments
*
*/
class SomeClass {

  /**
  * First line describing someMethod that is used in the index.
  * <p>
  * HTML formatted <!-- HTML Comment --> comments
  *
  * @param arg1 a description of the first parameter
  * @param arg2 a description of the second parameter
  * @return a description of the return value
  */
  int someMethod(int arg1, String arg2) {
    return 0;
  }
}
```

javadoc Example Output

Fig 2-6: **Javadoc output from example 2-3**



javadoc Comment Structure

The basic structure of a `javadoc` comment is:

```
/**
* first line.
* <p>
* comment body
* <p>
* more comment body
*
* @directive
* @directive
*/
```

javadoc Comment Structure

Some of the basic rules for `javadoc` are

- First line in each `javadoc` comment is used as an index entry
- Line ends at the first period
- Utilize HTML to make the `javadoc` generated documentation easier to read
- Paragraph tags <p> are used to start new paragraphs in the comment body
- Utilize directives
- The comment body ends when a directive is encountered
- Place `javadoc` comments before the class and each method in the class

javadoc Comment Structure

Directives are lines that

- Start with a `@`
- Are immediately followed by a keyword like
  - `param`
  - `return`
  - `author`
  - Etc.

javadoc Comment Structure

- For method comments
  - A `@param` directive is required for each parameter
  - Looks like

    ```
    @param nameofparamater  description which
    does on until another directive  or the
    end of the comment is encountered
    ```

- A `@return` directive is also required for methods, which describes what is returned and looks like

    ```
    @return a description of the return value
    of the method
    ```

Starting up the Application

- The `main(String [] args)` method has a special role in Java applications
- The `main` method contains the start-up code that is used to bootstrap the Java application
- The `main` method is the same as the mainline program in structured programming
  - The Java application runs as a process on the host operating system
  - `main` is the entry point to that process

Starting up the Application

- The `main` method works like this
  - `java HelloWorld` is executed at the command line
  - The class loader finds and loads the file `HelloWorld.class`
  - The JVM looks through the class file for a method that looks like

```
public static void main(String [] args) {
    /* main method body */
}
```

  - The JVM executes the body of the `main` method
  - When the `main` method finishes executing, the Java application finishes and the JVM exits

Starting up the Application

Example 2-4: Adding a main method to BankAccount

```
public class BankAccount {
  String accountNumber;
  int accountBalance;
  String accountType;
  String accountStatus;
  int queryBalance() {
    return 0;
  }
  int deposit (int amount) {
    return 0;
  }
  int withdraw(int amount) {
    return 0;
  }

  public static void main(String [] args) {
     System.out.println("Bank Account main method executing..");
  }
}
```

Use of main

- We create a special class which is responsible for starting up and shutting down the application
  - The `main` method then goes into this "application" class
  - Commonly called `Main.class`
- For example, we if we had a banking application we could
  - Define a new class called `BankApp`
  - Responsible for overall management of our banking application
- The `BankApp` class can start off like this

Example 2-5: The BankApp application class

```
public class BankApp {
  public static void main(String [] args) {
    System.out.println("Starting Bank Application…");
    // code will go here later
    System.out.println("Ending Bank Application…");
     }
}
```

Command Line Arguments

- Command line arguments are passed to an application following an the applications name
  `java HelloWorld arg1 arg2 "this is argument 3"`
- The command line arguments are passed to the `main` method as an array of `String`s

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

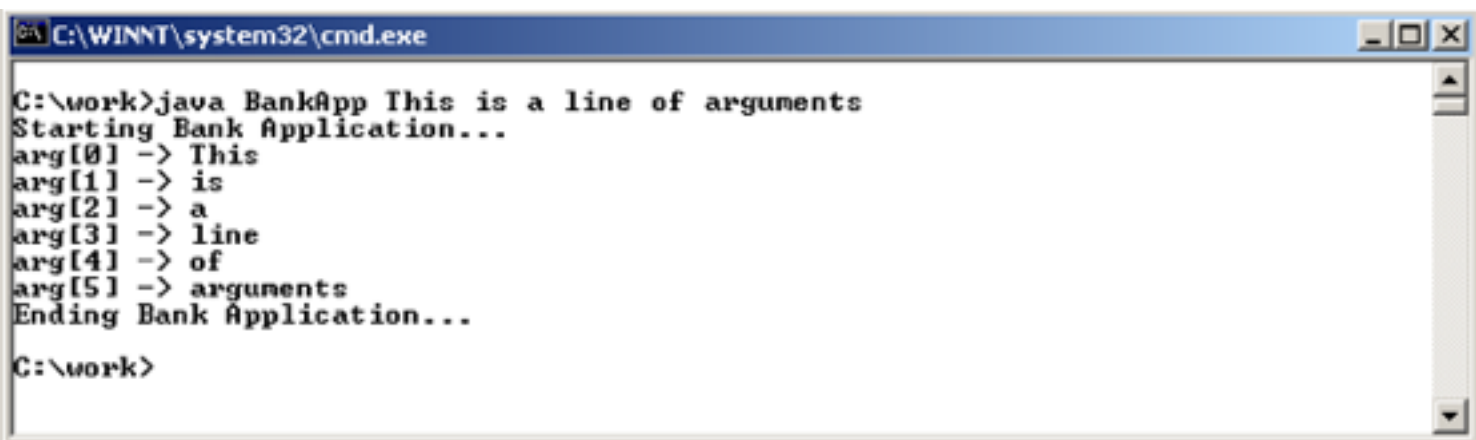
  - `args` is a variable name
    - It references an array of `String` objects
    - You can name it whatever you want
  - Has a zero length if no arguments were specified
  - The following example demonstrates this

Command Line Arguments Example

Example 2-6: Using command line arguments with main

```
public class BankApp {
  public static void main(String [] args) {
    System.out.println("Starting Bank Application");

    // print out the command line arguments
    for (int argcount = 0; argcount < args.length; argcount ++)
      System.out.println("arg["+argcount+"] -> "+args[argcount]);

    System.out.println("Ending Bank Application");
  }
}
```

Fig. 2-12: **Output of example 2-6**



The init method

- Applets do not have a `main` method
  - So how do they get bootstrapped?
  - This is the responsibility of the runtime container
- When a browser launches an `Applet` for the first time
  - Class definition for the `Applet` loaded into the local JVM
  - Browser calls the `public void init()` to initialize the `Applet`
  - Browser calls other *life-cycle* methods to manage the `Applet`
    - `public void start()`
    - `public void stop()`
    - `public void destroy()`

The Applet Lifecycle

Every programmer needs to implement the code for the four

lifecycle callback

- The `init` method - initializes the applet when it is first loaded by the browser
- The `start` method - sent from the browser to the applet to every time the browser returns to the page containing the applet
- The `stop` method - sent when the browser moves off the page containing the applet
- The `destroy` method - Sent when the browser shuts down

The Bank Application Class

- The `Applet` model is an example of good OO program design
- A runtime environment that
  - Creates the program objects
  - Initializes them
  - Kicks the application off
  - Shuts down the application gracefully
  - Disposes of the program objects
- In our banking application the application class will act like a runtime container

The Bank Application Class

We are going to use a `BankApp` class to fill three roles:

1. Manage our bank application's lifecycle
2. Act as a container for all of the objects that need to work together in our bank application
3. Provide services analogous to a runtime container for the objects that make up the bank application

Summary

We covered

- How a Java class definition is organized
- What `init` and `main` methods do in Java
- The purpose of an application class and a run-time container
- Using `javadoc` to create documentation

Variables, Operators and Data
(Chpt. 3 - Part 1)

Objectives

At the end of this module you should be able to:

- Describe the rules for creating legal variable names in Java
- Describe and use the basic primitive data types in Java
- Use `String` data
- Determine the data type of a literal

Strong Typing in Java

- Java is a strongly typed language
  - Each variable and each expression has a type
  - Can be identified by the compiler at compile time
  - A variable's type cannot be changed
- In loosely typed languages, like JavaScript & VB

Example 3-1: Loose variable typing in JavaScript

```
// JAVASCRIPT:  This is not allowed in JAVA!!!
// Declare a variable "x" with no type
var x
x = "Hi there"    // x is holding string data
x = 1234          // x is now holding numeric data
y = x + "343"     // String or numeric operation??
```

- Strong typing helps prevent errors

Data Types in Java

- There are two types
  - Reference data
  - Primitive data

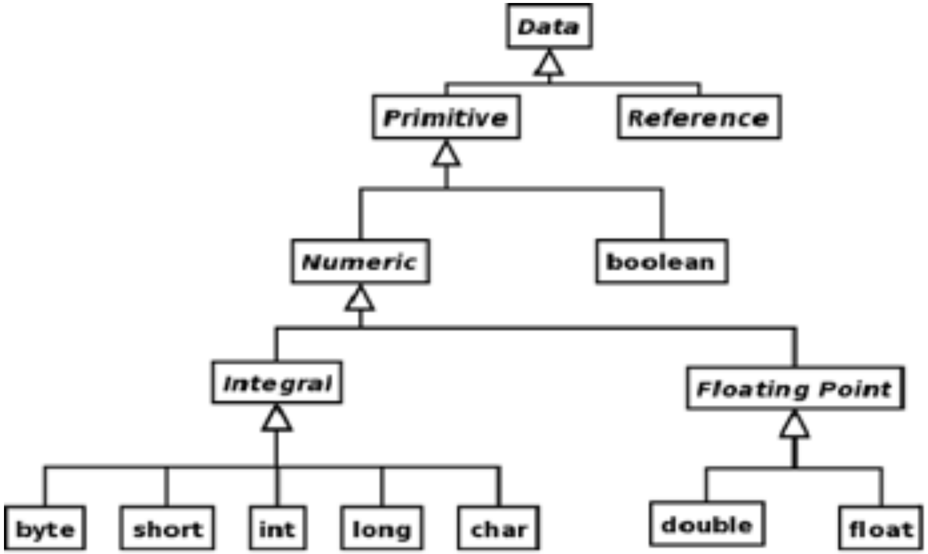


Fig. 3-1: **Hierarchy of data types in Java**

- The primitive data types resemble the types in C and C++
- Data types in Java are defined by the language specification
  - They are platform independent
  - For example, the data type `int` is *always* four bytes long

Identifiers

- Identifiers are used to describe
  - Classes
  - Variables
  - Method
- Identifier rules are platform independent
  - Must be an arbitrarily long sequence of letters and digits
  - Are case sensitive
  - The first character must be a letter
    - Any valid letter in the Unicode character set
    - The underscore "_" and dollar sign "$" are considered to be letters while symbols like © and + are not
  - They may **not** contain any white space
  - They may **not** be the same as reserved Java keywords

Reserved Keywords

Reserved Java keywords

`abstract`	`default`	`if`	`private`	`this`
`boolean`	`do`	`implements`	`protected`	`throw`
`break`	`double`	`import`	`public`	`throws`
`byte`	`else`	`instanceof`	`return`	`transient`
`case`	`extends`	`int`	`short`	`try`
`catch`	`final`	`interface`	`static`	`void`
`char`	`finally`	`long`	`strictfp`	`volatile`
`class`	`float`	`native`	`super`	`while`
`const`	`for`	`new`	`switch`	`true`
`continue`	`goto`	`package`	`synchronized`	
`false`	`null`			

Fig. 3-2: **Reserved Java Keywords**

`goto` and `const` are reserved but yet to be used

Variable Names

Example 3-2: Valid and invalid identifiers in Java

```
Account_Balance // valid - remember that _ is a letter
$34                 // valid - remember that $ is a letter
this      // invalid - same as reserved keyword
π                   // valid - Greek letter pi is a Unicode letter
This        // valid - different in case from 'this'
Next.item           // invalid - symbol "." not allowed
23skidoo            // invalid - must start with letter
```

Declaring a Variable

- Variables are declared in Java with the syntax

```
type name [= initial_value];
```

- It is good programming practice to *initialize variables when they are declared*
- A variable can also be initialized after it is declared
- Java will prevent the use of uninitialized variables in code

Declaring a Variable

Example 3-3: Initializing variables

```
String best = "Best"; // Preferred - initialized at declaration
String okToo;     // Declared - not initialized
int x;            // Declared - not initialized

okToo = "value";            // Now okToo is initialized.
x = x + 1;                  // ERROR! Use of an uninitialized variable
```

Declaring a Variable

Example 3-4: Initializing multiple variables

```
boolean a,b;                    // a and b are both of type boolean
boolean c = true, d = false; // initialization for both c and d
boolean e,f = true;      // WARNING! Only f is initialized!
```

Example 3-5: Variables in memory

```
int var1;
boolean var2 = true;
var1 = 9
```

Positioning Variable Declarations

- The basic principal in Java, and in OOP in general, is to declare a variable at its point of first usage

Example 3-6: Positioning variable declarations in code

```
class Test {
  public static void main(String [] args) {
    int sum = 0;
    for (int counter = 0; counter < 10; counter++)
      sum = sum + counter;
        String message = "The sum is ";
        System.out.println(message + sum);
    }
  }
}
```

Boolean Data Types

- `boolean` data is either `true` or `false`
- In C and C++, `boolean` values are numeric data
  - i.e. Zero is false, any non-zero is true
- In Java, `boolean` variables are not numeric
- Can have only the values `true` or `false`

Boolean Data Types

Example 3-7: boolean data in C+ and Java

```
// In C++ you do can this:
int x = 43;
// non-zero x is taken as a true
if (x) {
  System.out.println("x is "+x);
}
// In Java, the above code does not compile.  You need a
// boolean variable or expression.

boolean test = (x==43);
// boolean variable is OK
if (test) {
  System.out.println("x is "+x);
}
// OK because result of test is boolean
if (x == 43) {
  System.out.println("x is "+x);
}
```

Integral Data Types

- All integral values are signed - there are no unsigned values
- Integral values do not contain a decimal point

Type	Bytes	Minimum Value	Maximum Value
`byte`	1	-128	127
`short`	2	-32768	32767
`int`	4	-2147483648	214748364
`long`	8	-9223372036854775808	9223372036854775807

Fig 3-4: **Integral Data Types**

Integral Literals

- A sequence of digits without a decimal point is assumed to be integral data
  - literals are of type `int` unless there is an `L` - upper or lowercase - immediately following the digits
  - In this case, the literal is taken to be a `long`
  - `7836` and `-98` are `int`
  - `881L` and `-91121` are `long`
- Literals are interpreted in Base 10 unless
  - The literal starts with a `0`, it is interpreted as Base 8
  - The literal starts with a `0x` or `0X`, it is interpreted as Base 16

Integral Literals

Example 3-8 Integral literals

```
63     // an int in base 10
-63    // a negative int in base 10
63L    // a long in base 10
063    // an int in base 8 (equal to 51 in base 10)
063l   // a long in base 8
-063l  // a negative long in base 8

091     // illegal!  Cannot have the digit 9 in base 8!

0x33   // an int in base 16 (equivalent to 51 in base 10)
0X33L  // a long in base 16
0xFF   // an int in base 16
0xff   // same as the previous line - case does not matter.
0xg1   // illegal! Can only have a-f as base 16 digits.
-0xFF  // an negative int in base 16
```

Floating Point Data Types

- Floating point are numerical values with fractional parts.
- Two kinds of floating point numbers
- `float`: 4 bytes long
  - Largest `float` is 3.4028234 E +38
  - About 6 or 7 significant digits
- `double`: is 8 bytes long
  - Largest `double` in magnitude is 1.79769313486231570 E +308
  - About 15 significant digits

Infinities, Negative Zeros and Non-numbers

Example 3-9: Test class for positive infinity

```
class InfinityTest {
  public static void main(String [] args) {
       // Set up bigd as a large double
       double bigd = 1e306;

       // loop - we should see bigd overflow about the third iteration
       for (int i=1; (i<100) && (bigd<Double.POSITIVE_INFINITY); i++) {
         System.out.println("Iteration="+ i +": bigd="+bigd);
         bigd = bigd * 10.0;
       }//end for loop

  }//end main

}//end class
```

Floating Point Literals

Example 3-11: Floating point literals

```
38.0            // double
38.0f           // float
38.98D          // double
1.78e23         // double
1.78e23f        // float

-789.983        // double
-1.89e-17F      // float
```

Character Data

- A kind of integral data
  - The type name is `char`
  - Takes on values from 0 to 65535
- Java supports the Unicode standard - each character is stored as a two-byte representation
- ASCII is a subset of Unicode - Java handles the ASCII/Unicode conversions behind the scenes

Character Literals

- Character literals usually represent a single Unicode character in single quotes
- Character literals can also be the numeric code for Unicode characters
  - Unicode escape sequence notation
  - '`\udddd`' where `dddd` is the hexidecimal representation of the Unicode character
- Certain common non-printable characters, as well as the single and double quote and backslash, have special escape sequences that are recommended for use instead of the corresponding Unicode escape sequence

Character Literals

Example 3-12: Character literals

```
'a'          '7'          'ξ'          '©'          ' '
'\''         '\\'         '\u0F34'     '\u0004'     '\ffd1'
'_'

'ab'         // Not a char literal – two characters between quotes
'\ug189'     // Not a char literal – illegal Unicode code.
```

Fig: 3-5 **Unicode Escape Sequences**

```
'\b'         /* \u0008: backspace BS */
'\t'         /* \u0009: horizontal tab HT */
'\n'         /* \u000a: linefeed LF */
'\f'         /* \u000c: form feed FF */
'\r'         /* \u000d: carriage return CR */
'\"'         /* \u0022: double quote " */
'\''         /* \u0027: single quote ' */
'\\'         /* \u005c: backslash \ */
```

Strings

- There is no string primitive data type in Java
  - `String`s are actually a reference data type that is implemented in the Java SE APIs
  - Java allows `String` data to be used syntactically as if it were a primitive data type
  - Intended to make working with character strings more "programmer friendly"
- `String` literals are sequences of Unicode Characters
  - Enclosed in double quotes
  - `char` escape sequences are valid for `String`

Chars and Strings

Example 3-13: Using Character data

```
char a = 'a';          // single character
char b = 'b';          // single character
char nl = '\n';        // using the non-printable escape code for newline
char x = '\u7878';     // using the Unicode escape sequence
a + b;                 // the result is an int.
```

Example 3-14: Strings

```
String s = "This is a string";
s = "This is a string with a backspace \b in it";
s = "This is a string with a \" double quote inside";
String t = s;
t = "";         // this is the empty string - still a string though
t = 'a';        // illegal! 'a' is not a string.
```

Summary

We covered

- The rules for creating legal variable names in Java
- Describe and use the basic primitive data types in Java
- Use `String` data
- Determining the data type of a literal

Variables, Operators and Data
(Chpt. 3 - Part 2)

Objectives

At the end of this section you should be able to

- Describe what operators and expressions are
- Describe how operators are used to create expressions
- Describe the operators in Java, the kinds of data they operate on, and the types of expressions they produce
- Describe the difference between narrowing and widening operators
- Use the cast operator correctly

Operators and Expressions

- An expression in Java is something that evaluates to a result
  - A variable is an expression because it evaluates to a result - the value of the data it contains
  - A literal is also an expression
- An operator is used to combine two expressions to produce a new expression
- Think of variables as nouns and operators as verbs
  - Combine nouns and verbs to create phrases
  - Phrases correspond to expressions

Operators and Expressions (cont.)

- Every expression has a type, just like a variable
- The type of an expression is determined by the type associated with the data results when we evaluate the expression
- For example, a `boolean` expression is one that results in a `boolean` result while a `String` expression is one that results in a `String`

Operators

- Operators are of three valences in programming languages
  - Unary operators - operate on a single expression
  - Binary operators - combine two expressions
  - Ternary operators - combine three expressions
- Most operators fall in the binary operator category
- There is only one ternary operator in Java
- Java does not allow operator overloading like in C++ / C#

Operators - Example

There is no relationship between the type of operator (category)

and the expression type.

Example 3-15: Operators in Java

```
1 + 4  // arithmetic operator "+" operating on two ints
1.0 * 4.1     // arithmetic operator "*" operating on two doubles
true && false // logical operator operating on two booleans
true + false  // illegal! - you can't do arithmetic on booleans

// Error in the following line, even though almost all the
// operators in the expression are arithmetic, the final result
// is a boolean, and cannot be assigned to x.
int x = (((34 + 12)/13) * (89-16)/(13 *2)) > 0;
```

Arithmetic Operators

- These operators are the standard `+ - * / %` operators
  - Java also has the increment and decrement operators
  - Defined for both the integral and floating point types
- Mixed Mode Arithmetic
  - All arithmetic operators work on either two integral operands or two floating point operands
  - Mixed mode arithmetic means that one operand is integral and one operand is floating point
  - Then the integral operand is converted to a floating point number before the operation takes place

Mixed Mode Arithmetic

Converting from integral to floating point values may produce a

loss of precision that can show up at odd times. This is due to

rounding of floating point values

Example 3-16: Mixed Mode Arithmetic

```
1 + 4          // result is integral 5
1.1 + 4.2      // result is 5.3 (or 5.3000000000000001 sometimes)
1.0 + 4.0      // result is 5.0 - still floating point
1.0 + 4        // result is 5.0 - one operand is floating point
```

Division and Modulus

Example 3-17: Division in Java

```
17 / 3          // result is 5 - integral division
17 % 3          // result is 2 - remainder of 17 / 3
17.0 / 3.0      // result 5.2 - floating point division
17.3 / 3        // result is 5.766666666666667
                // -- floating point division
17.0 % 3        // result is 2.0 - floating point modulus
14.5 % 3.32     // result is 1.2200000000000006
                // whatever that means.
```

Increment and Decrement

- Java uses the C/C++ increment and decrement operator
  - `++`
  - `--`
- The are two forms
  - Postfix - `<VAR>++` and `<VAR>--`
  - Prefix - `++<VAR>` and `--<VAR>`
- In the postfix form the value of the variable is used in the expression first, then incremented or decremented
- In the prefix for the value of the variable is incremented or decremented first, then used in the expression

Increment and decrement operators in Java

```
x++      is equivalent to x = x + 1
++x      is equivalent to x = x + 1
x--      is equivalent to x = x - 1
--x      is equivalent to x = x - 1
```

Increment and Decrement Example

Example 3-18: Increment and Decrement

```
int i = 23;
double d = 0.0;

// Print out i
System.out.println("Value of i is "+ i);
// Print out ++i - i is printed out after being incremented
System.out.println("Value of ++i is "+ (++i));
// Print out i again - it has been incremented.
System.out.println("Value of i is "+ i);

// Print out i
System.out.println("Value of i is "+ i);
// Print out i++ - i is printed out before being incremented
System.out.println("Value of i++ is "+ (i++));
// Print out i again - it has been incremented.
System.out.println("Value of i is "+ i);
// Just to see that it works with floating points
System.out.println("Value of ++d is "+ (++d));
```

Increment and Decrement Example Output

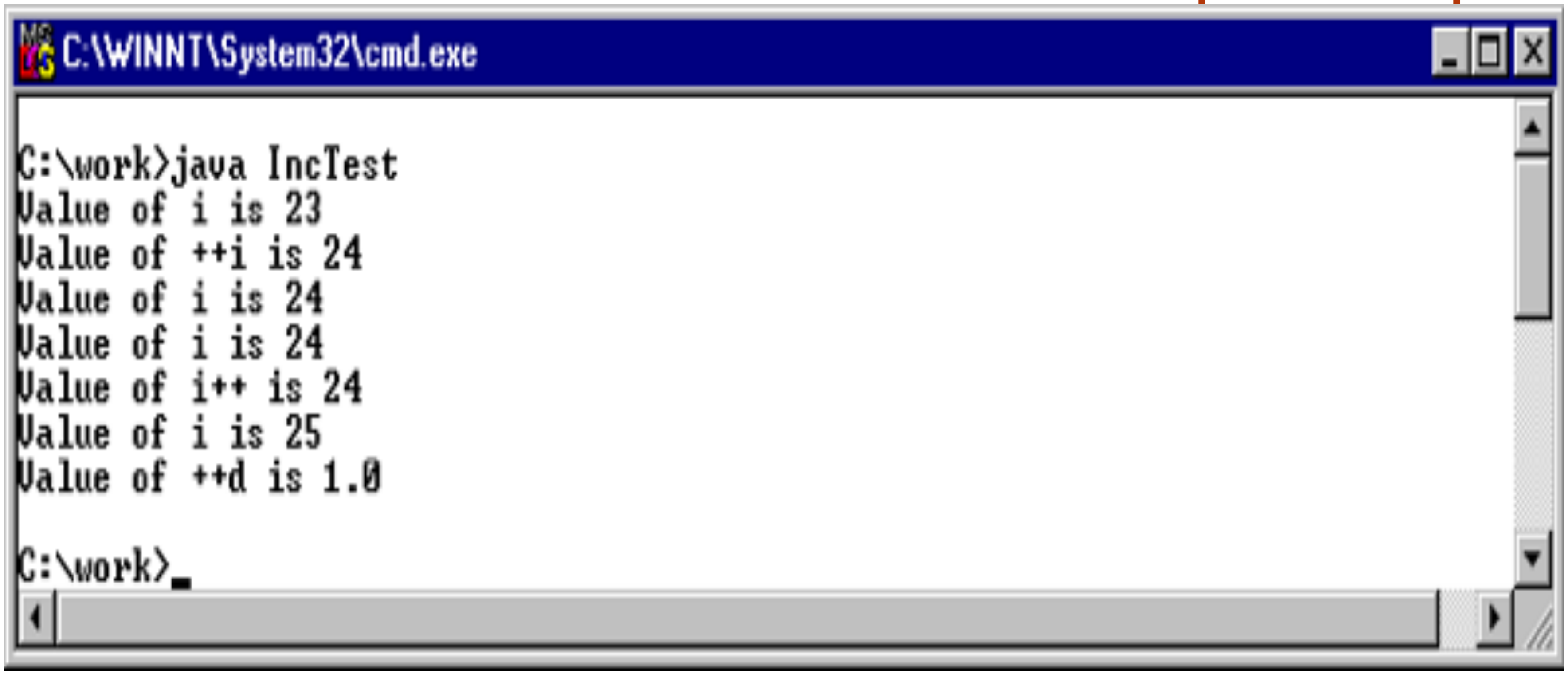


Fig. 3-5: **Output of the IncTest in example 3-18**

Comparison Operators With Numerics

- Comparison operators are defined for numeric and character data
- The result of all comparisons is either `true` or `false`

Comparison Operators in Java

`==`	Equality
`<`	Less than
`>`	Greater Than
`<=`	Less than or equal to
`>=`	Greater than or equal to
`!=`	Not equal

Comparison Operators With Other Types

- Only the `==` and `!=` operators are defined for `boolean` types
- Not all comparison operators work for `String` types
  - Remember, a `String` is not a primitive data type
  - Only the == and != work with `String` types, but not quite as you might expect
  - You will learn more about how these operators work in a later module

Cautions with Comparison Operators

- It is possible for a loss of precision to occur when working with floating point numbers
  - This is unavoidable
  - Sometimes using floating point numbers in comparisons can produce counterintuitive results

Example 3-19: Relational operators and floating point numbers

```
double d = 1.1 + 4.2; // As we saw, could be 5.3000000000000001
d == 5.3               // Because of rounding, this is false

double d1 = 1e300;    // d1 is a very large number
d1 < (d1 + 1)         // because of rounding, this is false
d1 == (d1 + 1)        // because of rounding, this is true
```

Logical Operators

- The operators `&`, `|`, `^`, and `!` all work according to the usual rules of `boolean` operations
- The two operators `&&` and `||` are called *short circuit* operators

Evaluation of Logical Operators
Assume x and y are boolean expressions.

```
x & y   true if both x and y are true, false otherwise
x | y   false if both x and y are false, true otherwise.
x ^ y   true if either x or y is true but not both
!x         false if x is true, true is x is false

x && y  same as &, but if x is false, y is not evaluated
x || y  same as | but if x is true, y is not evaluated.
```

Short Circuit Evaluations

- In a short circuit evaluation, we stop evaluating the expression as soon as we know what the outcome will be
- This avoids unnecessary processing if the first part of the evaluation is `false`

Example 3-20: Short circuit evaluation

```
int x = 0;
boolean test = false & (1 == ++x);
// x is incremented and is now 1
test = false && (2 == ++x);
// second operand is not evaluated!
// x still has value 1
```

Assignment Operators

Java does allow C/C++ style assignment operator notation

Example 3-21: Operator Assignment

```
x = x * 34;     //can be written as   x *= 34;
x = x / 2;      //can be written as   x /= 2;
x = x + y;      //can be written as   x += y;
X = x - y;      //can be written as    x -= y;
```

String Operators

- Operators in general are not defined for the `String` type data
- `String` catenation can be performed using
  - `+`
  - `+=`
- Only work when at least one operand is a `String`
- All other operands are converted to a `String`
- The two `String`s are then concatenated into a new `String`
- Any kind of data can be converted to a `String`

String Operators Example

Example 3-22: String Catenation

```
String message = "String data ";
int i = 34;
float f = 89.13F;
boolean b = true;
char c = '*';

message + i -> "String data 34"
f + message -> "89.13String data "
message + b -> "String data true"
message + c -> "String Data *"
f + " " + i -> "89.13 34"
b + " " + c + " " + i ->  "true * 34"

// implicit string conversion
(i + "") + f  -> "3489.13"
i + f -> 123.13
```

Operator Precedence and Associativity

Operator	Associates
[] . () function_call	Left to right
! ~ ++ -- cast new - +(unary form)	Right to left
* / %	Left to right
+ - (binary form)	Left to right
<< >> >>>	Left to right
< <= > >=	Left to right
== !=	Left to right
&	Left to right
^	Left to right
|	Left to right
&&	Left to right
||	Left to right
?:	Left to right
= (op=)	Right to left

Fig: 3-6: **Operator precedence in Java**

Operator Precedence and Associatively Example

Example 3-23: Operator Precedence

Since `*` has a higher precedence than `+`

```
2 * 4 + 3 -> 8 + 3 -> 11
3 + 2 * 4 -> 3 + 8 -> 11
```

But we can change the order of operations
with `()` to make the `+` be evaluated first

```
2 * (4 + 3) -> 2 * 7 -> 14
(3 + 2) * 4 -> 5 * 4 -> 20
```

The `&&` operator associates from left to right.
In the following assume `x` is 3, and `y` is 5

```
(x == 3) && (y == 5) && ( x == y) -> true && ( x == y ) -> false
```

which we can override with `()`

```
(x == 3) && ((y == 5) && ( x == y)) -> ( x == y) && false -> false
```

On the other hand assignment associates from right to left. Assume `y` is 5

```
x = y = 4 -> x = 4 -> 4
```

Widening Conversions

Java will always do a widening conversion

- Converting a numeric data type to a wider version of the same type
- The numeric value is preserved exactly without any loss in precision
- Converting any integral data type to a floating point type is allowed
- There is no loss of magnitude, but there can be a loss of precision

Widening Conversions Example I

Example 3-24: Widening Conversions

```
long longVar;
int  intVar;
short shortVar;
byte  byteVar;
char  charVar;
float floatVar;
double doubleVar;

byteVar = 120;
shortVar = byteVar;
intVar = shortVar;
longVar = intVar;
System.out.println("LongVar is "+ longVar); // value is 120L
```

Widening Conversions Example II

Example 3-25: Widening Conversions -- integral to floating point

```
long longVar;
float floatVar;
double doubleVar;

longVar = Long.MAX_VALUE;
floatVar = longVar;
doubleVar = longVar;
System.out.println("longVar is "+ longVar);
System.out.println("floatVar is "+ floatVar);
System.out.println("doubleVar is "+ doubleVar);

// Output is
longVar is 9223372036854775807
floatVar is 9.223372E18
doubleVar is 9.22337203685477 6E18
```

Narrowing Conversions and Casting

- Narrowing conversions are the opposite of widening conversions
  - Converting a data type to a smaller version of the same type
  - Converting from a floating point data type to an integral data type
- Java does *not* perform narrowing conversions automatically
- The data must be *cast* to the narrower type
  - A cast operator is represented as a new data type name in parentheses placed before the variable or expression to be cast

```
variable2 = (new_type) variable;
```

Narrowing Conversions and Casting Example

Example 3-26: Narrowing conversions

```
byte byteVar = (byte)255;
short shortVar = (short)214748360;
int intVar = (int) 1e20F;
int intVar2 =(int)Float.NaN;
float floatVar = (float)-1e300;
float floatVar2 = (float)1e-100;

System.out.println("(byte)255 -> " + byteVar);
System.out.println("(short)214748360-> "+ shortVar);
System.out.println("(int)1e20f -> " + intVar);
System.out.println("(int)NaN -> " + intVar2);
System.out.println("(float)-1e300 -> " + floatVar);
System.out.println("(float)1e-100 -> " + floatVar2);

// produces the output
(byte)255 -> -1
(short)214748360-> -13112
(int)1e20f -> 2147483647
(int)NaN -> 0
(float)-1e300 -> -Infinity
(float)1e-100 -> 0.0
```

String Conversions

- Any primitive data type can be converted to a `String`
  - By implicit `String` conversion
  - For each primitive data type, we can use a `toString()` function found in "wrapper" classes to convert the value to a `String`
    - Don't worry, you will learn more about this in a later chapter
- We can also convert from a `String` to various data types
  - This is more complicated because not every string of characters can be converted to a particular primitive data type
  - We will deal with handling this problem later

String Conversions Example

Example 3-27: Converting to and from strings

```
String s = "123";
String t;

int k = 123;
float f = 19.801F;

//Integer and Float are “wrapper” classes
t = Integer.toString(k);
t = Float.toString(f);

// This line will not compile, you can't convert from a
// String this way
k = (int)s;
```

Forbidden Conversions

- There are certain conversions that are never allowed in Java
- According to the Java language specification, these are
  - No conversion from any reference type to any primitive type
  - Except for the `String` conversions, no permitted conversion from any primitive type to any reference types
  - No permitted conversion to `boolean` types
  - No permitted conversion from `boolean` other to a `String` conversion

Summary

We Covered

- What operators and expressions are and how operators are used to create expressions
- The operators in Java, the kinds of data they operate on and what types of expressions they produce
- The difference between narrowing and widening operators
- Using the cast operator correctly

Control Structures in Java
(Chpt. 4 - Part 1)

Objectives

At the end of this module, you should be able to

- Describe what statements and blocks are
- Describe what a local variable is and its scope
- Describe the flow of control of a Java program
- Use `if` statements, `switch` statements

Statements and Blocks

- The sequence of program execution is controlled by statements
- Statements in Java can extend over any number of lines and are terminated by a semi-colon (`;`)
- Executable statements *only* exist inside method bodies
- Statements are often made up of expressions, but not always
- Expressions evaluate to a result, but statements don't have to – the `while` loop is a statement that is not an expression
- A statement can also be an expression
  - Called an expression statement
  - The result of the expression is discarded
  - e.g., the statement `x = 3;` returns and discards the value `3`

Syntax of Statements

Example 4-1: Expressions and statements

```
x = 7 + 1      // expression - not a statement
x = 7 + 1;     // adding a semi-colon makes this a statement.
int x;         // statement - not an expression
x++;           // expression and a statement

// a statement and complex expression
int y = x++ / ( z * 2.0);

// the empty statement - legal but pointless.
;

// following: a line containing three statements
x = 7 + 1; x++; System.out.println(x);

// following is one statement on three lines
x =
    7 +
        1;
```

Blocks

- A block is a sequence of statements enclosed in braces { }
- A block can occur anywhere in a Java program that a statement can
- Any statement in the block could itself be replaced by a block
- Nested blocks are allowed

Blocks and Nested Blocks Example

Example 4-2: Blocks and nested blocks

```
public class Ex4_2  {
  public static void main(String []args) {
    int x = 0;
       x++;
    int y = 0;
    { // Start of a user defined block
      { // Start of a nested user defined block
        System.out.println(x);
        System.out.println(y);
      } // End of a nested user defined block
      y = x % 3;
    } // End of the outer user defined block
    y++;
    {  // Start of a second user defined block
      System.out.println(x);
      System.out.println(y);
    } // End of the second user defined block
  } // End of the block that makes up the body of the main method
} // end of the block that makes up the class definition
```

Local Variables

- A local variable are defined within blocks
- Local variables are **not** automatically initialized
- Local variables only exist within the scope of the defining block
- Local variables can be used as temporary or “working” variables within the body of a block
- Local variables have a set lifetime, they
  - Come into existence when the flow of control passes through their declaration
  - Cease to exist when the flow of control passes out the defining block

Scope of a Local Variable Example I

Example 4-4: Scope of local variable `outerVar`

```
// scope of outerVar
void someMethod(int x) {
  System.out.println("Entering someMethod..");
  int outerVar = 1; //local to someMethod
  { //"inner block"
    System.out.println("Entering inner block…");
    int innerVar = 4; //local to "inner block"
    innerVar = outerVar + x;
    System.out.println("innerVar is "+ innerVar);
  }
  System.out.println("outerVar is "+ outerVar);
}
```

Scope of a Local Variable Example II

Example 4-5: Scope of local variable `innerVar`

```
// scope of innerVar
void someMethod(int x) {
  System.out.println("Entering someMethod..");
  int outerVar = 1; //local to someMethod
  { //"inner block"
    System.out.println("Entering inner block…");
    int innerVar = 4; //local to "inner block"
    innerVar = outerVar + x;
    System.out.println("innerVar is "+ innerVar);
  }
  System.out.println("outerVar is "+ outerVar);

}
```

Common Local Variable Errors Example

Example 4-6: Common local variable errors

```
// scope of innerVar
void someMethod(int x) {
  System.out.println("Entering someMethod..");
  int outerVar = 1;
  {
    System.out.println("Entering inner block…");
    // Error 1: referencing innerVar before it is declared
    System.out.println(innerVar);
    int innerVar = 4;
    innerVar = outerVar + x;
    System.out.println("innerVar is "+ innerVar);
    // Error 2: trying to declare a variable with a name
    // used by another local variable in the same scope
    int outerVar = 10;
  }
  System.out.println("outerVar is "+ outerVar);
  // Error 3: Trying to reference innerVar outside of its scope
  System.out.println(innerVar);
}
```

Basic if Statement Syntax

- The `if` statement looks like:

  `if (test-expression){ }`

- The test-expression is any expression that evaluates to one of the `boolean` values `true` or `false`
  - The expression must be contained in parentheses
  - The body of the `if` statement can be either a single statement or a block
  - If the body is a single statement, it must be terminated by a semi-colon

Basic if Statement Syntax Example

Example 4-7: Examples of if-then

```
// if statement with a single statement as a then-clause,
if (x == 3)
  System.out.println("x is, in fact, three");
x = 24;

// if statement with a body as a then-clause
boolean test = (x > 23);
if (test) {
  System.out.println("x is out of range");
  x = x - 10;
  System.out.println("value of x is reset to" + x);
}
x = 24;
```

if-else Statements

- The `if-else` form allows two mutually exclusive paths of execution
- The `if-else` form of the `if` statement looks like

```
if (test-expression) {
  then-clause
}
else {
  else-clause
}
```

- If the test expression is `true`, the then-clause executes exactly as we just saw in the previous section
  - If the test condition is `false`, the else-clause executes instead
  - Since the test-expression is `boolean`, one of the clauses will always execute

if-else Statements Example

Example 4-8: Examples of if-then-else

```
// if statement with a single statements in both then and else clauses.
if (x == 3)
  System.out.println("x is, in fact, three");
else
  System.out.println("x is NOT, in fact, three");
// if statement with a body in both then and else clauses
if (x > 23) {
  System.out.println("x is out of range");
  x = x - 10;
  System.out.println("value of x is reset to" + x);
}
else {
  System.out.println("x is in range");
  x++;
}
```

if-else Statements Example (cont.)

Example 4-8: Examples of if-then-else (continued)

```
// if statement with a body in then and a statement else clauses
if (x > 23) {
  System.out.println("x is out of range");
  x = x - 10;
  System.out.println("value of x is reset to" + x);
}
else
  System.out.println("x is in range");

// if statement with a statement in then and a body else clauses
if (test)
  System.out.println("x is out of range");
else {
  System.out.println("x is in range");
  x++;
}
```

The Dangling else Problem

Example 4-10: Dangling else

```
if (test1)
  if (test2)
    System.out.println("test1 and test2 true);
else
  System.out.println("test1 if false");

// what the programmer meant was
if (test1) {
  if (test2)
    System.out.println("test1 and test2 true);
}
else
  System.out.println("test1 if false");

// what the compiler saw was
if (test1) {
  if (test2)
    System.out.println("test1 and test2 true);
  else
    System.out.println("test1 if false");
}
```

if-elseif-else Statements

Example 4-11: Nested conditional -- multiple test values

```
int status = getStatus();
if (status == 0) {
  /* stuff to do if status is 0 */
}
else {
  if (status == 1) {
    /* stuff to do if status is 1 */
  }
  else {
    if (status == 2) {
      /* stuff to do if status is 2 */
    }
    else {
      /* stuff to do if status is anything else */
    }
  }
}
```

if-elseif-else Statements

- Most programming languages provide an alternate form to nested `if` statements
- In Java the syntax is

The if-else if-else construct

```
if (test1) first-then-clause
else if (test2) second-then-clause
else if (test3) third-then-clause
else else-clause /* the else clause is optional */
```

if-else if-else Example

Example 4-12: if-else if-else for example 4-11

```
int status = getStatus();
if (status == 0) {
  /* stuff to do if status is 0 */
}
else if (status == 1) {
  /* stuff to do if status is 1 */
}
else if (status == 2) {
  /* stuff to do if status is 2 */
}
else {
  /* stuff to do if status is anything else */
}
```

The switch Statement

- Sometimes, it is necessary to perform conditional behavior based on the differing numeric values

```
if(x == 2) //do something
else if(x == 3) // do something else
else if(x == 4) //do something else
else //do something default
```

- Using if, else if, else can be
  - Tedious (especially if you have many conditions)
  - Cause unwanted overhead (every each condition is evaluated until the right one is found)
- There is a control construct that helps with this - `switch`
- The `switch` construct only works on integers and characters

The switch Statement

The switch construct

```
// testvar is a variable of some type.
switch(testvar) {
  case value1:
    /* code to execute when testvar has the value value1 */
      break;
  case value2:
    /* code to execute when testvar has the value value2 */
      break;
  case value3:
    /* code to execute when testvar has the value value3 */
      break;

  /*--- more cases --*/
  case value_n:
    /* code to execute when testvar has the value value_n */
      break;
  default:
    /* code to execute when testvar none of the above values */
    break;
}
```

The switch Statement Example

Example 4-13: switch case example

```
char status = 'a';

switch (status) {
  case '*':
    System.out.println("Asterisk");
    break;
  case 'a':
    System.out.println("letter a");
    break;
  case 'z':
    System.out.println("letter z");
    break;
  default:
    System.out.println("Unrecognized character");
    break;
}
```

The switch Statement (cont.)

Example 4-13: switch case example (continued)

```
System.out.println("And continuing…");

status = 'g';
switch (status) {
  case '*':
    System.out.println("Asterisk");
    break;
  case 'a':
    System.out.println("letter a");
    break;
  case 'z':
    System.out.println("letter z");
    break;
  default:
    System.out.println("Unrecognized character");
    break;
}
System.out.println("And continuing…");
```

The switch Statement

- The ordering of the `case` statements is entirely up to the programmer
  - For example, the `default` case does not have to be the last `case` statement
  - For example, the order of `case` does not have to follow the logical ordering of integers
- The `default` case is optional
- Execution will continue into the next `case` statement if a `break` statement is not encountered. This called *falling-through*
- Fall through behavior allows the same code to applied to a set of test cases
- Applying falling-through with fancy you can achieve some advanced solutions

switch Statement Fall-through Example

Example 4-14: switch case fall through

```
char status = 'z';
switch (status) {
  case '*':
  case 'a':
    System.out.println("letter a or an asterisk");
    break;
  case 'z':
    System.out.println("letter z");
  case 'w':
    System.print("letter w");
        break;
  default:
    System.out.println("Unrecognized character");
    break;
}
```

The Ternary Operator

- There is one ternary operator in Java
  - Inherited from C and C++
  - The same as an `if-else` statement but the result is an expression
- The conditional operator is `? :`

```
test-expression ? then-expression : else-expression
```

- The test-expression must be a `boolean` expression
  - If test-expression evaluates to `true`, the then-expression is evaluated and the result returned
  - If the test-expression is `false`, then the else-expression is evaluated and returned

The Ternary Operator Example

Example 4-15: Conditional operator

```
class Ex4_15 {
  public static void main(String [] args) {
    String message;
    boolean test = false;
         // Using an if statement
    if (test){
      message= "Then clause";
    }
       else {
     message = "Else clause";
    }
    System.out.println(message);
      // Using a conditional operator
      System.out.println(true ? "Then expression" :
                                   "Else expression");
      System.out.println(false ? "Then expression" :
                                   "Else expression");
  }
}
```

Summary

We covered

- What statements and blocks are in Java
- What a local variable is and its scope is
- What the flow of control in Java program is
- `if` statements and `switch` statements

Control Structures in Java
(Chpt. 4 - Part 2)

Objectives

At the end of this section, you should be able to:

- Use `while`, `do-while` and `for` loops
- Use `break` and `continue` statements
- Use method overloading correctly

Loops in Java

- There are three looping structures in Java
  - `while` loops
  - `do-while` loops
  - `for` loops
- The `while` and `do-while` loops are more natural for iterating while some test condition is `true`
- The `for` loops are more natural when iterating over arrays or when it is convenient to have loop counter or index available
- The syntax for should be familiar

while

- The most basic looping structure
  - Condition is evaluated prior to executing the body
  - The body is executed as long as a condition is true
  - The `while` loop looks like this:

```
while(test-condition) {
  loop-body
}
```

- Remember test conditions must result in either a `boolean` `true` or `false`

while Example

Example 4-16: `while` loop example

```
int count = 0;

// while loop with a statement as a loop body
while (count < 10)
  System.out.println("count is "+ count++);

// while loop with a block as a loop body
count = 0;
while (count < 10) {
    System.out.println("count is "+ count++);
    count++;
}
```

do-while

- A variation on the `while` loop is the `do-while` loop
  - Always executes the body of the loop at least once
  - Determines subsequent execution based on condition
  - The `do-while` loop looks like this:

```
do {
  loop-body
}while (test-condition);
```

- The `do` indicates the start of the loop body and the test condition appears after the `while`
  - Notice that there is a semi-colon after the test condition
  - If the loop-body is a statement it must be terminated with a semi-colon

do-while Example

Example 4-18: while loops of Example 4-16 as do-while loops

```
int count = 0
// do-while loop with a statement as a loop body
do
  System.out.println("count is "+ count++);
while (count < 10);
// do-while loop with a block as a loop body
count = 0;
do {
  System.out.println("count is "+ count);
  count++;
}while (count < 10);
```

Command-line Input Example I

Example 4-19: Using a `while` loop

```
class DoWhileTest1 {
    public static void main(String [] args) {
      // Using a while loop
      try {
        char input;
        String output = "";
        input = (char)System.in.read();
        while(input != '\n') {
          output = input + output;
          input = (char)System.in.read();
        }
        System.out.println(output);
      }
      catch (Exception e) {
        System.out.println("IO Exception:" + e);
      }
    }
}
```

Command-line Input Example II

Example 4-20: Using a `do-while` loop

```
class DoWhileTest2 {
  public static void main(String [] args) {
      //Using a do-while loop
      try {
        char input;
        String output = "";
        do {
          input = (char)System.in.read();
          output = input + output;
        } while(input != '\n');
        System.out.println(output);
      }
      catch (Exception e) {
        System.out.println("IO Exception:" + e);
      }
   }
}
```

Arrays

- An array is a data structure that holds multiple values of the same type
  - The values of an array are called the array *elements*
  - They are accessed by index or their numerical position from the start of the array
  - In Java, all arrays are zero-based which means that the index of the first position is 0
  - Initialized arrays have an intrinsic attribute describing the size - `length`
- The easiest way declare and initialize an array:

```
data_type [] array_name = { list, of, initial,
  values };
```

Creating Arrays Example

Example 4-21: Creating arrays

```
class ArrayTest {
  public static void main(String [] args) {
    int [] bob = {9,78,-3,0,89 };
    String [] a = {"black", "brown", "white",
                    "green", "blue", "brown"};
  }
}
```

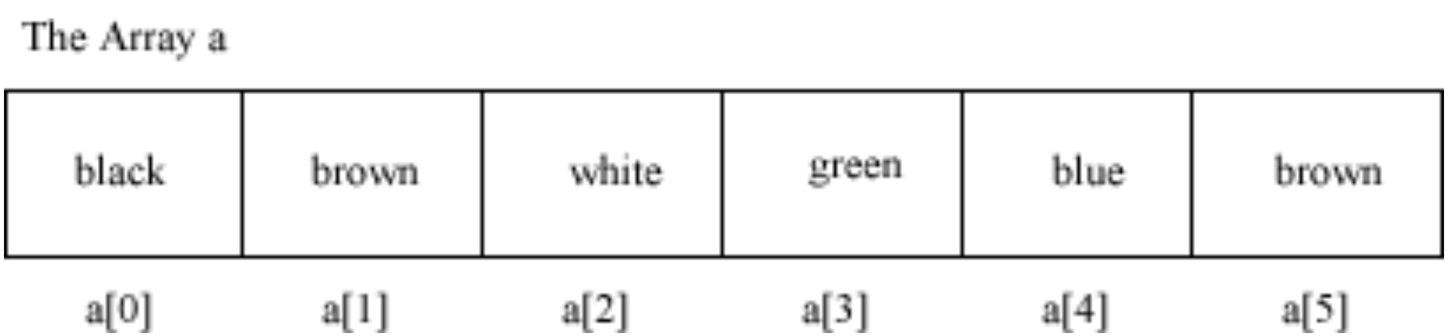


Fig. 4-1: **Array from example 4-21**

Using Arrays Example

Example 4-22: Using arrays

```
class ArrayTest2 {
  public static void main(String [] args) {
    int [] bob = {9,78,-3,0,89 };
    String [] a = {"black", "brown", "white", "green",
                                         "blue", "brown"};
    int index = 0;
    while (index < a.length) {
      System.out.println("a["+index+"] ->"+a[index]);
      index++;
    }
    index = 0;
    while (index < bob.length) {
      if (index == 3 || index == 2) bob[index]=9999;
      System.out.println("bob["+index+"] ->"+bob[index]);
      index++;
    }
  }
}
```

for Loops

- Provide the same functionality as `while` loops
- Are intended to make iterations involving counters or indexes easier to program
- Has the following structure

```
for (initial-clause; test-clause; iteration-clause)
  loop-body
```

- It is not required that you provide a valid expression for each clause, we will see more on this later

for Example I

Example 4-23: Basic for loop

```
class ForLoop {
  public static void main(String [] args) {
    int sum = 0;
    int index;
    for (index = 1; index <=100; index++) {
      sum += index;
    }
    System.out.println("Sum ="+sum);
  }
}
```

for Example II

Example 4-24: More for loop

```
class ForLoop2 {
  public static void main(String [] args) {
    int [] forwards = {0,1,2,3,4,5,6,7,8,9};
    int [] backwards = {0,1,2,3,4,5,6,7,8,9};
    int idx1, idx2;
    for (idx1 = 0, idx2 = 9; idx1 <forwards.length; idx1++, idx2--){
      backwards[idx2]=forwards[idx1];
    }
    System.out.println("Backwards is now..");
    for (idx1 = 0; idx1 <backwards.length; idx1++){
         System.out.println(backwards[idx1]);
    }
  }
}
```

Local Variables

- Remember, local variables are defined within blocks
- Their existence is defined by their scope
- Local variables can be used in loops to hold temporary data
- Local variables can be defined as part of the loop test expression or as variables in the body

Local Variables in for Loops

Example 4-25: More for loop

```
class ForLoop3 {
  public static void main(String [] args) {
    int outer = 0; //local to main
    //middle is defined in the for construct
    for (int middle = 0; middle <10; middle++) {
      int inner = 0; //same scope as middle
      inner++;
      outer++;
      System.out.println("outer="+outer+" middle="+middle+
              " inner="+inner);
    }
    System.out.println(" outer="+outer);
    // This following two lines will not compile because
    // they are out of scope for middle and inner.
    //System.out.println(" inner="+inner);
    //System.out.println(" middle="+middle);
  }
}
```

Local Variables Example Output

```
C:\WINNT\system32\cmd.exe

C:\work>java ForLoop
outer=1 middle=0 inner=1
outer=2 middle=1 inner=1
outer=3 middle=2 inner=1
outer=4 middle=3 inner=1
outer=5 middle=4 inner=1
outer=6 middle=5 inner=1
outer=7 middle=6 inner=1
outer=8 middle=7 inner=1
outer=9 middle=8 inner=1
outer=10 middle=9 inner=1
 outer=10

C:\work>_
```

Fig. 4-2: **Output of example 4-25**

More on Local Variables in Loops

Placement of local variables can sometimes cause unwanted results

```
// This is okay
int k; String s;
for (k=23, s="hi";;;)
{ /*body */ }

// As is this
for (int k=0, j=1;;)
{/* body */ }

// This doesn't work -- compiler thinks j is being redeclared.
int j;
for (int k =0, j=1;;)
{ /* body */ }

// Nor does this -- compiler does not expect to find "String"
// type must be consistent across all declared variables
for (int k=0, String s="hi";;) {/* body */}
```

Variations on the for Loop

The Java language does not require evaluating expressions for the different clauses in a `for` loop

Example 4-26: Variations on the `for` loop

```
// Variation one - an infinite for loop
for (;;) {
   // Only way to end this one is with a break
   if (conditions) break;
}
// Variation two - only a test expression
// Equivalent to a while loop
int k = 0;
for (; k < 10;) {
  k++
}
// Variation three - just an initialization clause
int z;
for (int k = 0, z=35;;) {
  if (++j >0) break;
}
// Variation four - just an iteration clause
int z=35;
for (;;z++) {
  if (z >100) break;
}
```

The break Statement

- The preceding flow control constructs executed until some condition fails
- In some cases, it is necessary to stop the execution of the loop structure due to some “local” condition
- The `break` statement in Java produces an abrupt termination of control
- As soon as the `break` statement is encountered then the processing breaks out of the loop and goes to the first statement after the loop body
- It is possible to utilize nested breaks within nested loops

The break Statement

Example 4-27: The break statement

```
class BreakTest {
  public static void main(String [] args) {
    int [] values = {9,45,1,0,98,102,-34};
    int idx = 0;
    while (idx < values.length) {
      if (values[idx ] == 98)
        break; //break out of while
      idx++;
    }
    if (idx == values.length) {
      System.out.println("98 was not found");
    }
    else {
      System.out.println("98 found at index "+ idx +" in values");
    }
  }
}
```

Nested loop break Example

Example 4-28: The break statement in nested loops

```
class BreakTest2 {
  public static void main(String [] args) {
    int [] target = {9,45,1,0,98,102,-34};
    int [] test =  { 9, 30, 102, 14 };
    int idx1 = 0;
    int found = -1;
    // Outer loop
    while (idx1 < test.length) {
      int idx2 = 0;
      while (idx2 < target.length) {// Inner loop
        if (test[idx1] == target[idx2])  {
          found = idx2;
          break; //break inner loop
        }
        idx2++;
      }
      idx1++;
    }
    if (found == -1) {
      System.out.println("No test values found");
    }
    else {
      System.out.println("Found test value "+ target[found] +
        " at index "+ found +" in target");
    }
  }
}
```

Labels

- When dealing with nested loops, using break may not provide the level of termination precision required
- For example, you may want to break out of the entire looping structure when something fatal occurs
- Java provides a supporting construct called *labels*
- Anything in Java can be labeled; however labels really only make sense in the context of loops
- The basic syntax of a label is
  `label_name:statement`
- Labels used with `break`s tell the JVM specifically where to terminate

The break Statement

Example 4-29: The labeled break statement in nested loops -- this works

```
class BreakTest3 {
  public static void main(String [] args) {
    int [] target = {9,45,1,0,98,102,-34};
    int [] test =  { 9, 30, 102, 14 };
    int idx1 = 0;
    int found = -1;
    // Outer loop with label zippy
    zippy: while (idx1 < test.length) {
      int idx2 = 0;
        // Inner loop
        while (idx2 < target.length) {
          if (test[idx1] == target[idx2])  {
            found = idx2;
            break zippy; //stop the execution of zippy:while
          }
          idx2++;
        }
        idx1++;
     }
     if (found == -1) {
         System.out.println("No test values found");
     }
     else {
       System.out.println("Found test value "+ target[found] +
            " at index "+ found +" in target");
     }
  }
}
```

The continue Statement

- In some cases, breaking out of a loop may not be desired
- Instead of breaking out of the loop, the current iteration is cancelled and the next iteration is started
- Just like `break` statements, `continue` statements can be labeled or unlabeled

continue Statement Example I

- In example 4-30, if the remainder after division by 2 (the modulus operator) is not 0, then we have an odd number so we just start the next iteration and skip over the output statement

Example 4-30: The continue statement

```
class ContinueTest {
  public static void main(String [] args) {
    int [] target = {9,45,1,0,98,102,-34};
    for (int idx = 0; idx < target.length; idx++) {
      if (target[idx] %2 != 0) {
        continue;
      }
      System.out.println(target[idx]+" is even");
    }
  }
}
```

continue Statement Example II

Example 4-31: The labeled continue statement in nested loops

```
class ContinueTest {
  public static void main(String [] args) {
    int [] target = {9,45,1,0,98,102,-34};
    int [] test =  { 9, 30, 102, 14 };
    // Outer loop with label zippy
    zippy: for (int idx1= 0; idx1 < test.length; idx1++) {
      for (int idx2 = 0; idx2 < target.length; idx2++) {
        if (test[idx1] == target[idx2])  {
          System.out.println("Found "+ test[idx1] +
                 " at " + idx2);
          continue zippy;
        }
      }
    }
  }
}
```

Methods

- Methods are equivalent to functions in structured programming.
- A method consists of three parts: a return value, a signature and a body.
- Methods define the behaviors of the Objects created from the class templates

Example 4-32: Some methods

```
class Test {
  int method1() {/* method body */}
  void method2(int x) {}
  void method2(float x) {}
}
```

Return Values of Methods

- All Methods have a return value
- In Java, a method can have only a single return value
- There are three common types of return values:
  - `void` - nothing is returned
  - Primitive - `int`, `char`, `long`, etc.
  - Reference Value (object) - `String`, `BankAccount`, etc

Example 4-32: Some methods

```
class Test {
  int method1() {/* method body */}
  void method2(int x) {}
  void method2(float x) {}
}
```

Method Signatures and Overloading

- The signature of a method is the method name (indentifier) plus the list of parameter types
- All method signatures in a class must be unique, which means that all methods either have:
  - Different names or
  - The same name, but different argument lists
- Methods the same name but differing in argument lists are referred to as *overloaded* methods
- The uniqueness of signatures only applies *within* a class definition

Method Signatures and Overloading

Example 4-34: Classes and methods

```
class Bob {
  static void print(int x) {
    System.out.println("Integer: "+ x);
  }
  static void print(float x) {
    System.out.println("Float: "+ x);
  }

class Fred {
  static void print(int x) {
     System.out.println("Integer: "+ x);
  }
}
```

Method Signatures and Overloading

A look at `java.io.PrintStream` would reveal overloading of `println`

```
void println()              Write line separator string.
void println(boolean x)     Print a boolean
void println(char x)        Print a character.
void println(char[] x)      Print an array of characters.
void println(double x)      Print a double.
void println(float x)       Print a float.
void println(int x)         Print an integer.
void println(long x)        Print a long.
void println(Object x)      Print an Object.
void println(String x)      Print a String
```

Method Invocation

- In order to perform some operation, a method invocation must occur
- In fact, an initial method invocation is required in order for our application to execute - main
- In the definition of applications, objects will interact with other objects using method invocation
- In Java, parameters declared in methods have scope local to the method
- Currently, Java does not support optional parameters or default parameters to methods

Calling a Method

Example 4-35: Calling a method

```
class Ex4_35 {
   static void print(int x) {
     System.out.println("Integer: "+ x);
   }
   // illegal because the method has the same signature as another
   // method in this class - only differs by return value.
   // static int void print(int x) { return x;}
   public static void main(String [] args) {
     int z = 3;
     print(z);
   }
 }
```

Method Parameters

Example 4-36: Argument promotion

```
class Ex4_36 {
  static void print(int x) {
    System.out.println("Integer: "+ x);
  }
  static void print(byte x) {
    System.out.println("Byte: "+ x);
  }
  public static void main(String [] args) {
    int z = 3;
    print(z);
    byte b = 9;
    print(b);
    short s = 12;
    print(s);
  }
}
```

Method Parameters

Fig 4-4: **Output of example 4-36**

Optional Arguments and Default Parameters

Example 4-37: Default Parameter

```
class Ex4_37 {
  static void print(int x, char language) {
    switch (language) {
      case 'E':
      case 'e':
        System.out.println("The number is: "+ x);
        break;
      case 'F':
      case 'f':
        System.out.println("Le numeral est:"+ x);
        break;
         /* -- more cases here -- */
     }
     return;
   }
   // The version where language defaults to English
   static void print(int x) {
     print(x,'E');
     return;
   }
   public static void main(String [] args) {
     print(3);
     print(4,'e');
     print(5,'f');

   }
}
```

Returning From A Method

- A method ends when it encounters a `return` statement
- A `return` statement usually has a type following it
- The return type matches what is specified in the method declaration
- If a method is declared to return `void`, no `return` statement is needed

Summary

We covered

- `while`, `do-while` and `for` loops
- `break` and `continue` statements
- method overloading correctly

Objects And Classes
(Chpt. 5 - Part 1)

Objectives

At the end of this module, you should be able to

- Use the `new` operator to create objects
- Describe how reference variables work
- Use instance variables and methods
- Discuss the use of constructors

Creating Objects

- Every OOP language must have a mechanism for creating objects from the class definitions
- Java uses the instantiation mechanism found in other OOP languages -- the `new` operator
- There is only one way to create an object in Java - by using the `new` operator
- The `new` operator is used in conjunction with a *constructor* to create, *instantiate*, an object
- The virtual machine is responsible for creating the memory associated with the object and initializes the memory through a constructor

Creating a BankApp Object

Example 5-1: Creating a BankApp object

```
public class BankApp {
  public static void main(String [] args) {
     // create the BankApp object
     BankApp thisApp = new BankApp();
  }
}
```

Sequence of Instantiation

A lot goes on behind the scenes when you create a new object

1. The JVM determines what type of object to create
   1. Looks at the *type* following the new operator
   2. We will look at constructors in detail a bit later
2. The JVM loads the associated class (if it is not already loaded)
3. The JVM allocates enough memory in the *Heap* to hold the newly created object – think *“garbage collector”*
4. The new object is initialized by
   1. Performing default initialization of all instance variables
   2. Executing the specified constructor
5. A *reference*, which we can think of as a pointer to a newly created object, is then returned and assigned to the *reference variable* `thisApp`

Reference Variables

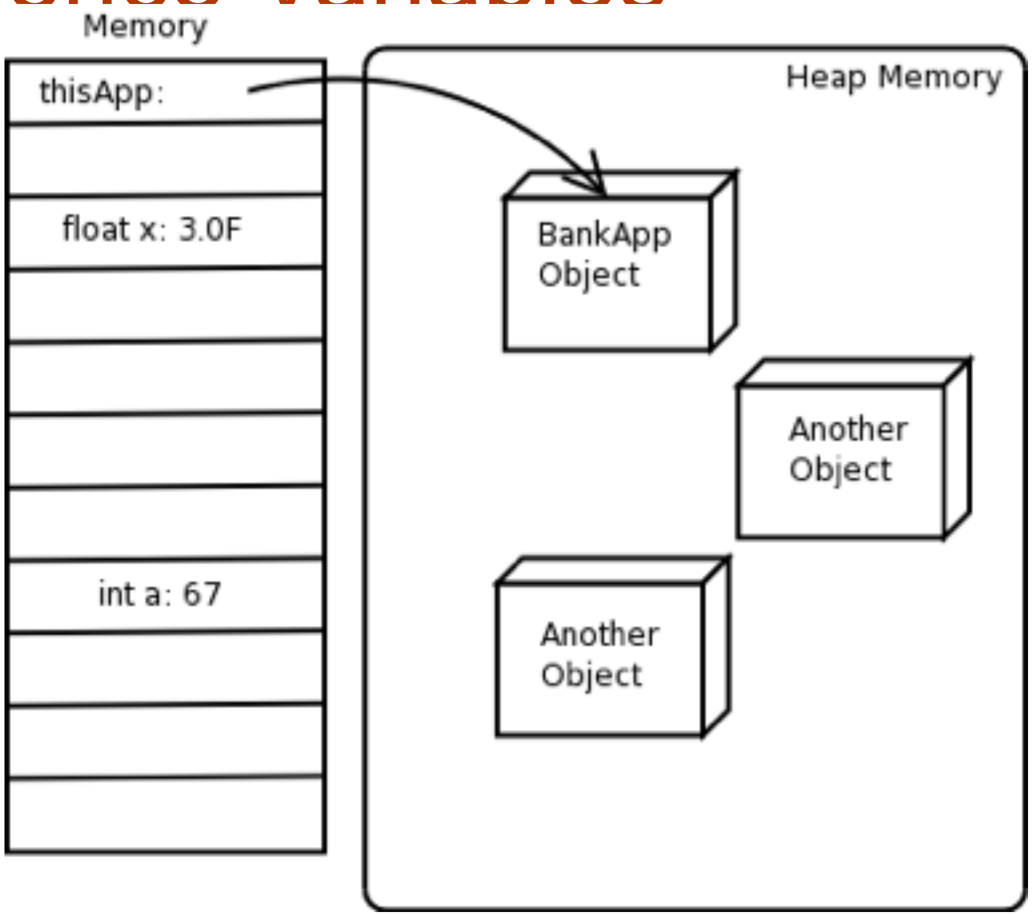


Fig. 5-1: **Memory allocation for reference variable thisApp**

Reference Variables

Example 5-2: Creating a BankApp object and null

```
public class BankApp {
  public static void main(String [] args){
    // Declare the variable.
    BankApp  thisApp = null;
    // Create and assign the object
    thisApp = new BankApp();
    // Create another BankApp object
    // don't assign IT to a variable
    // now we have no way to refer to it!
    new BankApp();
  }
}
```

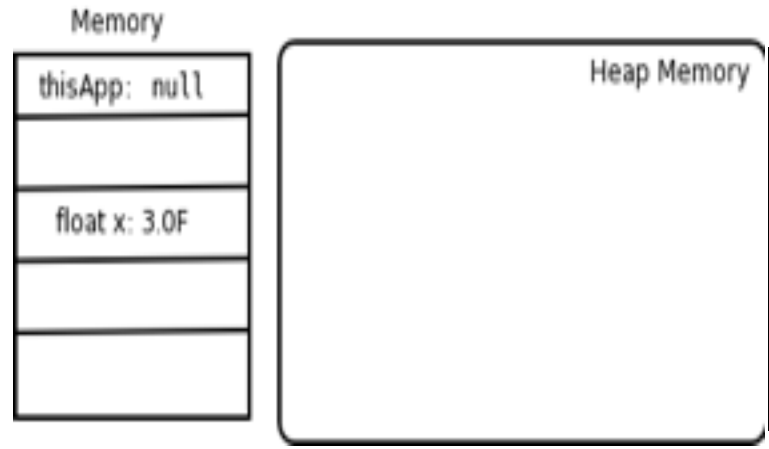


Fig. 5-2: **Reference variable with no associated object**

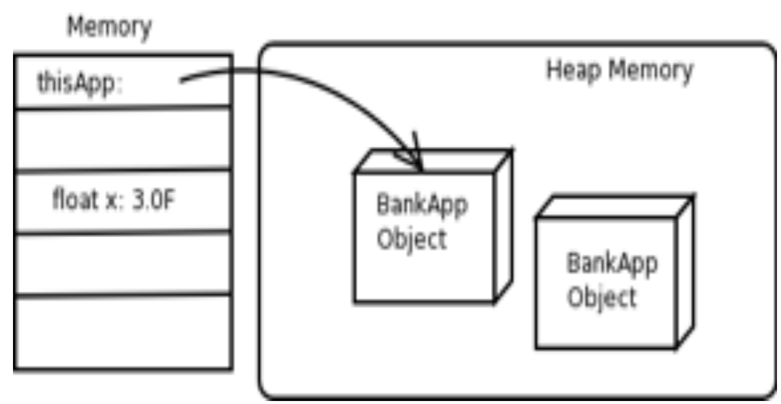


Fig 5-3: **Final state of example 5-2**

Object Description

- Objects are normally described by two things
  - Instance variables (*state*)
  - Instance methods (*behavior*)
- Both instance variables and methods are defined in class
- Their availability for use occurs once an object has been instantiated
- Referred to using the *dot-notation*

Object Description (cont.)

- Instance variables are known by many names
  - *Attributes*
  - *States*
  - *Instance Variables*
- Though they each technically have a different meaning, the names are commonly used interchangeably Do not exist until an object of that type is instantiated
- Instance methods are also known by many names
  - *Behaviors*
  - *Methods*
  - *Instance Methods*
- Though they each technically have a different meaning, the names are commonly used interchangeably Do not exist until an object of that type is instantiated

Instance Variables

- Hold data for a specific object
  - Each object has its own memory for the instance variables
  - Instance variables exist as long as the containing object exists
  - Instance variables live and die with their instance
- Can be either primitive data types or reference types
- Instance variables are initialized two ways:
  - Through default initialization performed by the JVM
  - Through constructor-defined initialization
  - Constructor-defined initialization always occurs after the default initialization process

Instance Variables (cont.)

- The instance variable values can be adjusted either by
  - Accessing them directly
    `objectVariable.variableName = xxx;`
  - Invoking a method that manipulates them
    `objectVariable.setVariableName(xxx);`
- The manner in which you access instance variables will depend on class design

Instance Variable Example

Example 5-3: Using an instance variable

```
class BankAccount {
  float balance;
  String accountNumber;
}
class BankApp {
  public static void main(String [] args) {
    System.out.println("Starting banking application");
    // Create a couple of bank accounts.
    BankAccount account1 = new BankAccount();
    BankAccount account2 = new BankAccount();
    // Print out the objects!!
    System.out.println("Account1 =" + account1);
    System.out.println("Account2 ="+ account2);
    // Set the balances and account numbers
    account1.balance= 34.50F;
    account2.balance = 100.00F;
    account1.accountNumber= "888888";
    account2.accountNumber = "337722";
    // Print out the data
    System.out.println("Account 1:");
    System.out.println("Account Number:"+
    account1.accountNumber + "  Balance=" + account1.balance);
    System.out.println("Account 2:");
    System.out.println("Account Number:"+
    account2.accountNumber + "  Balance=" + account2.balance);
  }
}
```

Instance Variable Example Output

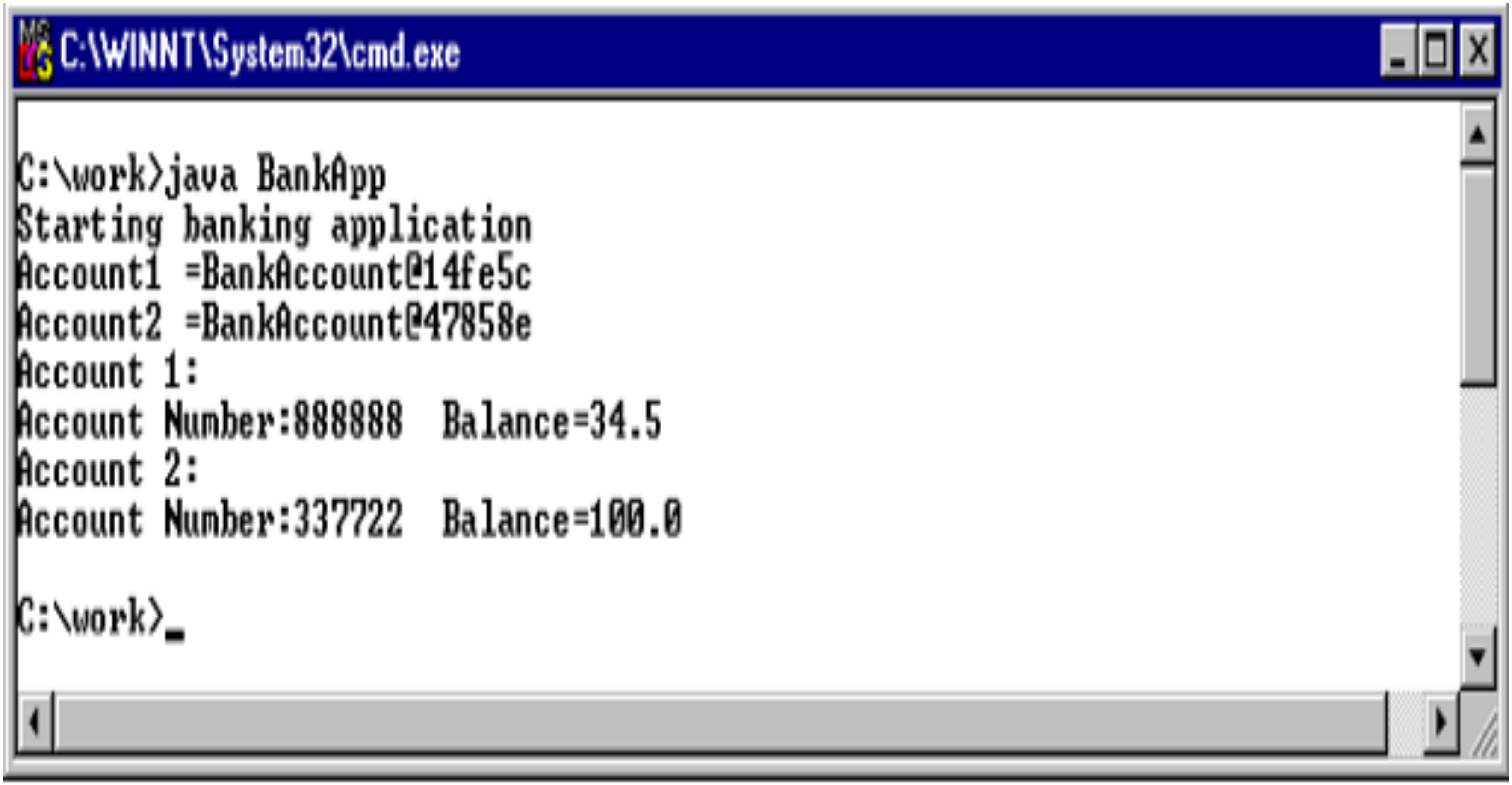
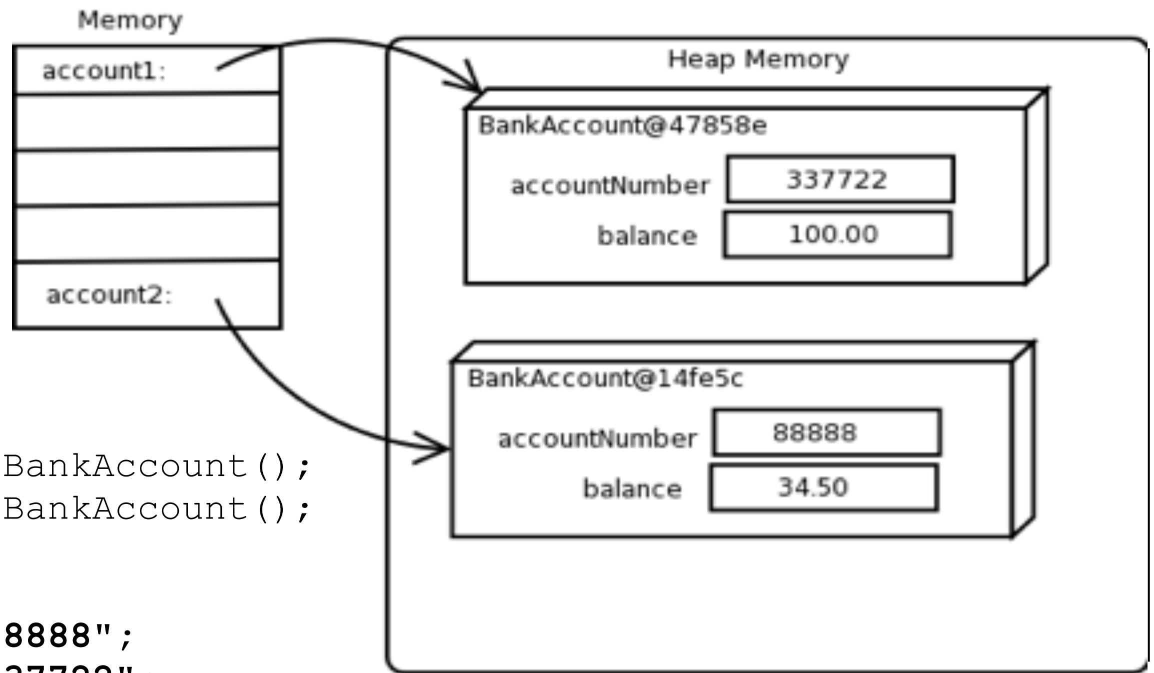


Fig. 5-4: **Output from example 5-3**

Referencing Instance Variables




```
BankAccount account1 = new BankAccount();
BankAccount account2 = new BankAccount();
account1.balance= 34.50F;
account2.balance = 100.00F;
account1.accountNumber= "888888";
account2.accountNumber = "337722";
```

***Fig: 5-5*: Objects at the end of example 5.3**

Instance Methods

- Perform some functionality on the object
- Commonly associated with underlying instance variables
- Typically instance methods fall into two categories when dealing with instance variables
  - Accessors – retrieve values
  - Mutators – change values
- Instance methods are “initialized” as part of the object class loading
  - Unlike instance variables, instance methods are shared by all instances of a specific class
  - When an instance method is invoked, it is invoked on a specific object
  - Sharing the definitions lowers the memory footprint
  - You should not be too concerned about this

Instance Methods

- The instance methods are invoked using the dot-notation

```
objectVariable.setVariableName(xxx);
objectVariable.methodName(arg1,arg2..);
```

- Remember, since they are associated with an instance, the instance must exist before calling a method
- Instance methods follow the method syntax we discussed earlier

```
 <access_modifier> <return> <identifier>(<parameter list>)
float deposit(float amt)
```

  - We will cover access modifiers later

Instance Method Example

Example 5-5: Invoking instance methods

```
class BankAccount {
  float balance;
  String accountNumber;

  float queryBalance()
  {
    return balance;
  }

  float deposit(float amt) {
    balance = balance + amt;
    return balance;
  }

  float withdraw (float amt) {
    balance = balance - amt;
    return balance;
  }
}
```

Instance Method Example (cont.)

Example 5-5: Invoking instance methods (continued)

```
class BankApp {
  public static void main(String [] args) {
    System.out.println("Starting banking application");
    // create two new bank accounts.
    BankAccount account1 = new BankAccount();
    BankAccount account2 = new BankAccount();
    // Make deposits into each
    account1.deposit(40.00F);
    account2.deposit(231.98F);
    // Display their balances
    System.out.println("Account1 balance: " + account1.queryBalance());
    System.out.println("Account2 balance: " + account2.queryBalance());
    // Make a withdrawal from each account
    account1.withdraw(10.00F);
    account2.withdraw(1000.00F);
    // Display their balances
    System.out.println("Account1 balance: " + account1.queryBalance());
    System.out.println("Account2 balance: " + account2.queryBalance());
  }
}
```

Instance Method Example Output

```
C:\WINNT\system32\cmd.exe

C:\work>java BankApp
Starting banking application
Account1 balance: 40.0
Account2 balance: 231.98
Account1 balance: 30.0
Account2 balance: -768.02
Ending Bank App...

C:\work>_
```

Fig. 5-7: **Output from example 5.5**

Application Correctness

- Two standards of correctness
  - Program correctness
  - Application correctness
- Program correctness defines
  - The best way to write code
  - Goes beyond just style, considers reusability, performance, robustness, etc
  - Often referred to as "best practices"
- Application correctness means
  - Application adheres to business rules for the problem domain
  - The business rules are implemented properly in the code

Business Rules

- Policies, procedures, and workflows automated by the application are commonly referred to as business rules
  - It is the programmer's responsibility to incorporate the business rules
  - It is important to ensure objects in the application conform to those rules
  - Business rules may take be represented as logic rules
- Business rules can also place execution and environment constraints on an application
- Hopefully, it is not the programmer's job to figure out the business rules
  - Domain experts are responsible for defining and deriving the rules
  - In our case, the domain experts are the banking staff at   bank

Business Rules

- Typically when methods are invoked, the method "consults" business rules when performing the operation
- Sometimes these business rules are called *guards*
  - They guard against a method executing when it should not
  - There are two kinds of guards that we generally deal with in OOP
    - **Preconditions**: **`boolean`** conditions that must be **`true`** before the method can be executed
    - **Postcondiitions**: **`boolean`** conditions that must be **`true`** after a method executes

Business Rules and Methods

Example 5-6: Adding guards to instance methods

```
class BankAccount {
  float balance;
  String accountNumber;
  int accountStatus;
  float queryBalance() {
    // This is a problem for now
    if (accountStatus != 0) {
      return 0.0F;
    }
    return balance;
  }
  float deposit(float amt) {
     // This is a problem for now
     if (accountStatus != 0) {
        return 0.0F;
     }
     balance = balance + amt;
     return balance;
  }
  float withdraw (float amt) {
     // This is a problem for now
     if (accountStatus != 0) {
         return 0.0F;
     }
     // Do the withdrawal only if no overdraft results.
     if (amt <= balance) {
         balance = balance - amt;
     }
     return balance;
   }
 }
```

Initialization of Instance Variables

- Variables must be initialized before they can be used
  - This rule is strictly enforced with local variables
  - This rule is relaxed with instance variables
    - This provides flexibility –
    - You can defer the initialization of an instance variable until the constructor is invoked
- Three options for instance variable initialization
  1. Allow the instance variable to be initialized by default
     - Performed automatically
     - Referred to as “default initialization”
  2. Initialize a variable when we declare it in the class definition
     - Performed as a result of assignment
     - Referred to as “explicit initialization”
  3. Initialize a variable in the constructor
     - Useful when initialization requires applying business rules
     - Referred to as “constructor initialization”

Initialization of Instance Variables (cont.)

- Initialization mechanism have different results
  - Default initialization
    - Reference variables, including `String` variables, are all initialized to `null`
    - Numeric values are initialized to the appropriate zero value and
    - `boolean` types are initialized to `false`
  - Explicit initialization
    - Variables initialized to some specific value
    - Default initialization does not occur
  - Constructor initialization
    - Occurs after default / explicit
    - Overrides initialization performed in (1) and (2)
    - Requires explicit logic
- Normally, instance variables are initialized in the constructor

Explicit Initialization Example

Example 5-7: Initialization of instance variables

```
class BankAccount {
  float balance = -1.0F;
  String accountNumber = "NotSet";
  int accountStatus = -1;
  char accountType = " "
}
```

Constructors

- Objects are creating through a `new SomeType()` call
- After the memory has been created, the object is initialized in the constructor
- Constructors are basically methods
  - Have a special purpose
  - Follow some certain rules

Constructor Purpose

- The purpose of the constructor is to initialize the newly created
  - Prevents instance variables from incorrect initialization
  - Any other initialization or startup code can be executed
  - Perform complex initialization logic that can not be done as an explicit initialization
- Because proper object initialization is so important, the Java compiler will add a constructor automatically for you
  - Referred to as the default constructor
  - Think of it as performing default initialization for your object
  - Compiler doesn’t know much about your class, so don’t expect anything fancy in the default constructor

Constructor Rules

Constructors must abide by some specific rules

- The constructor *always*
  - Has the same name as the class
  - Remember Java is case sensitive
- Constructors can be overloaded
  - Similar to method overloading
  - May be multiple constructors with different argument lists
- Does not declare a return value
  - This is not the same as returning `void`
  - A constructor actually returns a reference to the newly created object

Constructor Example

Example 5-8: Constructors for `BankAccount` class

```
class BankAccount {
  float balance = -1.0F;
  String accountNumber = "NotSet";
  int accountStatus = -1;
  char accountType = " ";

  BankAccount(String num, char type) {
    accountNumber = num;
    accountType = type;
    balance = 0.0F;
    accountStatus = (type == 'p')? 100: 0;
  }
  BankAccount(String num, char type, float bal) {
    accountNumber = num;
    accountType = type;
    balance = bal;
    accountStatus = (type == 'p')? 100: 0;
  }
  BankAccount(String num) {
    accountNumber = num;
  }
  /* -- rest of class -- */
}
```

Proper Constructor Form

- Typically a class defines multiple constructors
  - Each constructor varies by argument list
  - Though the constructors are different, they should perform the same level of initialization
- Having many constructors
  - Provides flexibility
  - Can be error prone if done wrong
- Constructors should refer to other constructors
  - To minimize redundant code
  - Provide centralized initialization
  - Simplify maintenance

Proper Constructor Form (cont.)

- When referring to other constructors
  - Utilize a built-in mechanism - `this(…)`
  - Think of `this(…)` as constructor calling another constructor
    - Like a method call
    - JVM determines which constructor to call
- Use `this(…)`
  - As the first line in your constructor
  - Can perform other operations once `this(…)` “returns”

Proper Constructor Form

Example 5-9: Constructors for `BankAccount` class

```
class BankAccount {
  float balance = -1.0F;
  String accountNumber = "NotSet";
  int accountStatus = -1;
  char accountType = " ";

  BankAccount(String num, char type, float bal) {
    accountNumber = num;
    accountType = type;
    balance = bal;
    accountStatus = (type == 'p')? 100: 0;
  }
  BankAccount(String num, char type) {
    this(num,type, 0.0F);
  }
  BankAccount(String num) {
    this (num, 'p');
  }
  /* -- rest of class -- */
}
```

The Default Constructor

In the first module, we used the disassembler (`javap`) to look into our `HelloWorld` class

Reproduced Figure 1-13

The Default Constructor

Example 5-10: Default constructors

```
//This compile and runs
class Test1 {
  int x;
  public static void main(String [] args) {
    Test1 t = new Test1();  // this is the default constructor
    System.out.println(t);
  }
}

//This does not compile
class Test2 {
  int x;
  // Adding this constructor prevents the default
  // costructor is not provided
  Test2(int xs) {
    x = xs;
  }
  public static void main(String [] args) {
    Test2 t = new Test2();  // this constructor no longer exists.
    System.out.println(t);
  }
}
```

Summary

We covered

- Using the `new` operator to create objects
- Describing how reference variables work
- Using instance variables and methods
- Describing and using constructors

Objects And Classes
(Chpt. 5 - Part 2)

Objectives

At the end of this section, you should be able to:

- Describe the use of the `public` and `private` access modifiers
- Use class methods and variables
- Use the `final` keyword with variables
- Use `String` and `StringBuffer` objects
- Use arrays
- Use wrapper classes

Object Reference Semantics

- Objects are accessed using references
- References are variables contains a “pointer” to an object
  - A 32-bit integer containing the heap location of your object
  - The reference value is hidden from you
- Copying a reference value only copies the reference value
  - You do not actually copy the underlying object
  - The end result is two references with the same value, referring to the same object in the heap

Object Reference Semantics Example

Example 5-11: Reference variable assignment

```
class BankApp {
  public static void main(String [] args) {
    // Declare the reference variables
    BankApp thisApp = null;
    Bankapp anotherApp = null;
     // Create the object and assign the reference value
    thisApp = new BankApp();
    // Now assignment of reference variables
    anotherApp = thisApp;
  }
}
```

Object Reference Semantics

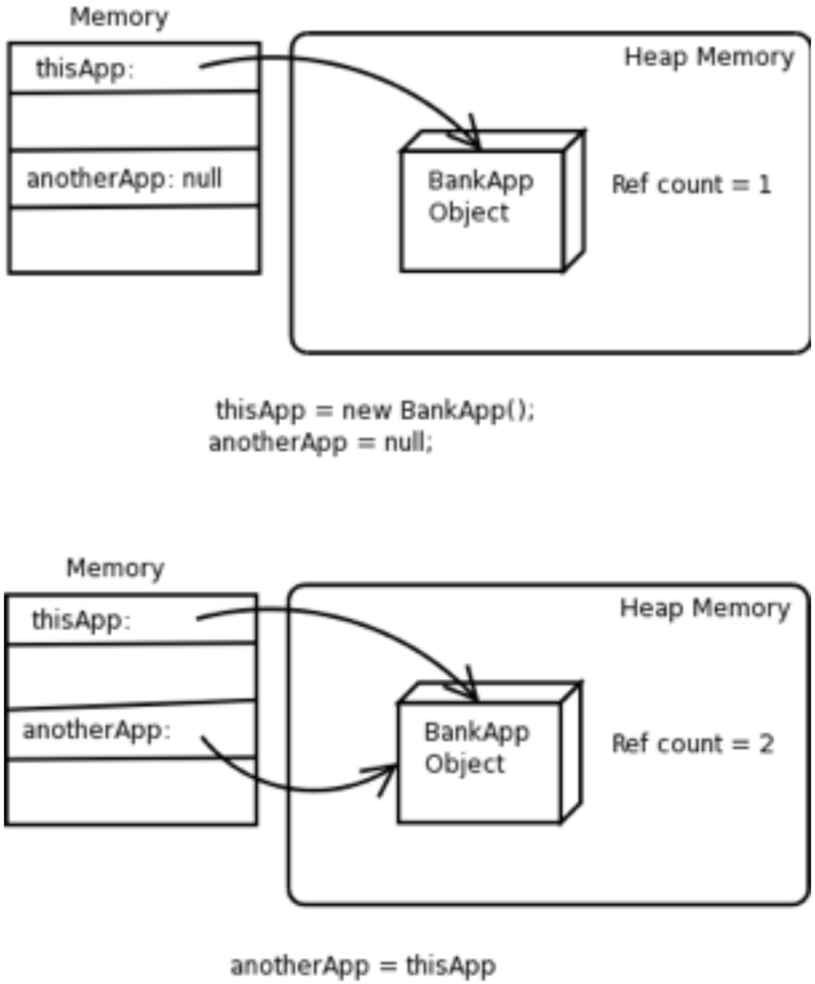


Figure 5-8: **Reference Semantics of a variable assignment.**

Object Life Cycle & Garbage Collection

- Java provides built in memory management
  - Used for allocating memory
  - Used for de-allocating memory – a.k.a. garbage collector
- The garbage collector (*gc*) is a daemon (background) thread that runs in the virtual machine
- There are many different types of garbage collection algorithms; in general the gc check on a regular basis which objects have become moribund

Object Life Cycle & Garbage Collection (cont.)

- The garbage collector frees up occupied but not referred to memory automatically
  - An object is in scope as long as there is at least one reference to it
  - As soon as the reference count of an object hits 0, then the object is moribund or ready to die
- Objects can go out of scope when
  - Their reference variables become `null` (through assignment)
  - Their reference variables are reassigned with a new value
  - Their reference variables go out of scope (local reference variables)

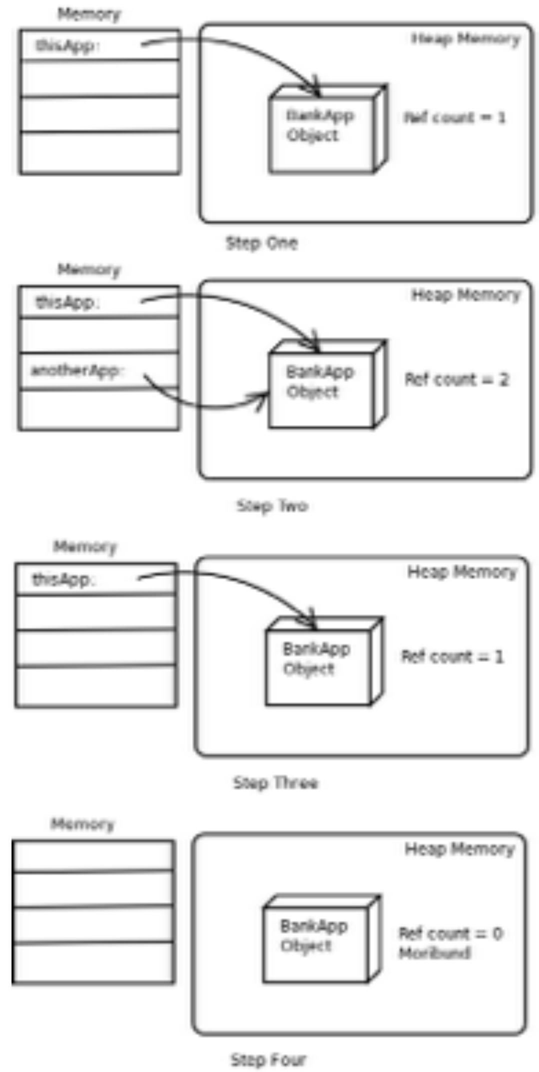
Object Life Cycle & Garbage Collection Example

Example 5-12: Life cycle of an object

```
class BankApp {
  public static void main(String [] args) {
    // thisApp is now a local variable to this block
    {
      BankApp thisApp = new BankApp();
      // BankApp object now has a reference count of
      // Step one in fig. 5-9
      {
        BankApp anotherApp = thisApp;
        // BankApp object now has a reference count of
        // Step two in fig. 5-9
      }
      // anotherApp is out of scope. Reference count is now 1
      // Step three in fig. 5-9
    }
    // thisApp is now out of scope, Reference count is 0
    // BankApp object is moribund waiting for garbage collection
    // Step four in fig. 5-9
  }
}
```

Object Life Cycle & Garbage Collection

```
{
  BankApp thisApp = new BankApp();



      {
        BankApp anotherApp = thisApp;








      }






 }
```



The finalize() Method

- Before an object is garbage collected, the JVM system calls its `finalize()` method
- The `finalize()` method gives the object a chance to return allocated or in-use system resources
  - Kind of the opposite of a constructor
  - There is no guarantee that the `finalize()` method will actually be called
  - The original intent of the `finalize()` method was to free up memory that was allocated through the JNI API, by C and C++ methods
- You can include a finalize method in your class

```
protected void finalize()
```

finalize() Method Example

Example 5-13: A finalize method

```
class BankApp {
  BankApp() {
    System.out.println("Creating BankApp");
  }
  protected void finalize() throws Throwable {
    System.out.println("Finalizing BankApp");
    return;
  }
  public static void main(String [] args) {
    new BankApp();
  }
}
```

Access Modifiers and Encapsulation

- Object Oriented Analysis and Design encourages the use of encapsulation
- Encapsulation is defined as data hiding
  - Think of encapsulation as a black box
    - Hide the “dirty” or sensitive details of objects
    - Prevents misuse
    - Thwarts “hacking”
  - Encapsulation should be applied to objects
- Objects are logical containers of
    - Data or state
    - Functionality or behavior

Access Modifiers and Encapsulation (cont.)

- An object's variables should be not be exposed to other objects
  - Instance variable exposure can allow direct variable access
  - This would circumvent any business rules you have in place
  - Would also allow object to obtain and corrupt sensitive data
- Sensitive and critical behaviors should also be hidden from other objects

Access Modifiers and Encapsulation (cont.)

In Java, we use *access modifiers* to create encapsulation

- Access modifiers define a level of accessibility for classes
  - Class variables
  - Class methods
- Access modifiers define a level of accessibility for instances
  - Instance variables
  - Instance methods
  - Constructors
- Basic syntax is:
  - Variables

```
<access_modifier> Type identifier;
```

  - Methods

```
<access_modifier> <return_type> identifier(<parameter list>)
```

Access Modifiers and Encapsulation (cont.)

- Java has four access modifiers
  - `private` – only the class and object can access
  - *default* – only class, object, and subclass in same library (*package*)
  - `protected` – the class, object and subclasses in any package
  - `public` – any class, object, subclass
- We will cover access modifiers in more detail when we discuss packages
- The default access modifier is automatically added if an access modifier is not explicitly specified

Private Variable Example

Example 5-14: Access Modifiers

```
class BankAccount {
    // Instance Variable
    private float balance;
    // Constructor
    BankAccount(){
      balance = 0.0F;
    }
    // Instance Methods
    float queryBalance() {
      return balance;
    }
    float withdraw(float amt) {
      if ((amt > 0.0F) && (amt <= balance)) {
        balance -= amt;
      }
     return balance;
    }
    float deposit(float amt) {
      if (amt > 0.0F) {
        balance += amt;
      }
     return balance;
    }
 }
```

Private Variable Example (cont.)

Example 5-15: Access Modifiers

```
class BankApp {
  public static void main(String [] args) {
    System.out.println("Starting banking application...");
    // Create a bank account
    BankAccount act = new BankAccount();
    act.deposit(100.00F);
    System.out.println("Balance is " + act.queryBalance());
    // This following line will not compile !!
    System.out.println("Balance is "+ act.balance);
    System.out.println("Ending banking application...");
  }
}
```

```
//producers the compiler error output
BankApp.java:9: balance has private access in BankAccount
System.out.println("Balance is "+ act.balance);
                                      ^

1 error
```

Private Variable Example

Example 5-16: Access Modifiers

```
class BankAccount {
  // Instance Variables
  String accountNumber = null;
  char accountType;
  float balance;
  int accountStatus;
  // Instance Methods
  float queryBalance() {
    return balance;
  }
  float withdraw(float amt) {
    if ((amt > 0.0F) && (amt <= balance)) {
      if ((accountType == 's' && accountStatus == 100) ||
          (accountType == 'c' && accountStatus == 0)) {
        balance -= amt;
      }
    }
    return balance;
  }
. . .
```

Private Variable Example (cont.)

Example 5-16: Access Modifiers (continued)

```
    float deposit(float amt) {
      if (amt > 0.0F) {
        if ((accountType == 's' && accountStatus == 100) ||
                (accountType == 'c' && accountStatus == 0)) {
          balance += amt;
        }
      }
      return balance;
    }
}
```

Private Method Example

Example 5-17: Access Modifiers

```
class BankAccount {
  // Instance Variables
  String accountNumber = null;
  char accountType;
  float balance;
  int accountStatus;
  // Private Instance Methods
  private boolean isAccountOK() {
    return ((accountType == 's' && accountStatus == 100) ||
            (accountType == 'c' && accountStatus == 0));
  }
  // Instance Methods
  float queryBalance() {
    return balance;
  }
. . .
```

Private Method Example

Example 5-17: Access Modifiers (continued)

```
float withdraw(float amt) {
  if ((amt > 0.0F) && (amt <= balance)) {
    if (isAccountOK()) {
      balance -= amt;
    }
  }
  return balance;
}
float deposit(float amt) {
  if (amt > 0.0F) {
    if (isAccountOK()){
      balance += amt;
    }
  }
  return balance;
}
}
```

Private Variable Example

Example 5-18: Private access between object of the same type

```
class A {
  private int var = 0;
  void changeVar( A otherA) {
    otherA.var++;
  }
  public static void main(String [] args) {
    // create an two A objects
    A firstA = new A();
    A secondA = new A();
    // use the first A object to change the private data in
    // the second A object
    firstA.changeVar(secondA);
  }
}
```

Public Method Example

Example 5-19: Public Access

```
class BankAccount {
  // Instance Variables
  String accountNumber = null;
  char accountType;
  private float balance;
  int accountStatus;
  // Private Instance Methods
  private boolean isAccountOK() {
    return ((accountType == 's' && accountStatus == 100) ||
             (accountType == 'c' && accountStatus == 0));
  }
  // Public Instance Methods
  public float queryBalance() {
    return balance;
  }
}
. . .
```

Public Method Example (cont.)

Example 5-19: Public Access (continued)

```
    public float withdraw(float amt) {
      if ((amt > 0.0F) && (amt <= balance)) {
        if (isAccountOK()) {
          balance -= amt;
        }
      }
      return balance;
    }
    public float deposit(float amt) {
      if (amt > 0.0F) {
        if (isAccountOK())balance += amt;
      }
      return balance;}
    }
}
```

Class Variables and Methods

- Java provides a mechanism to declare variables that belong to the class as a whole and not to a specific object
  - Called `static` or class variables
  - Provide much of the functionality that was provided by global variables in structured languages
  - `static` variables belong to the class and are shared by all instances of the class
- Think of class or `static` variables as being variables that are global within a class
- Static variables can be accessed by either instance methods or static methods
- Use the dot-notation to access `static` variables

```
class_name.staticVariableName
BankAccount.NumberOfAccounts
```

Class Variables

- To make an instance variable `static`, you simply place the keyword `static` before the variable definition
- For example, the following defines a `static` data member and initializes it:

Example 5-20: Static Variables

```
class BankAccount {
  // Class Variables
  static int NumberOfAccounts = 0;

  // Instance Variables
  String accountNumber = null;
  char accountType;
  private float balance;
  int accountStatus;
  // rest of class defintion
}
```

Referencing Static Variables

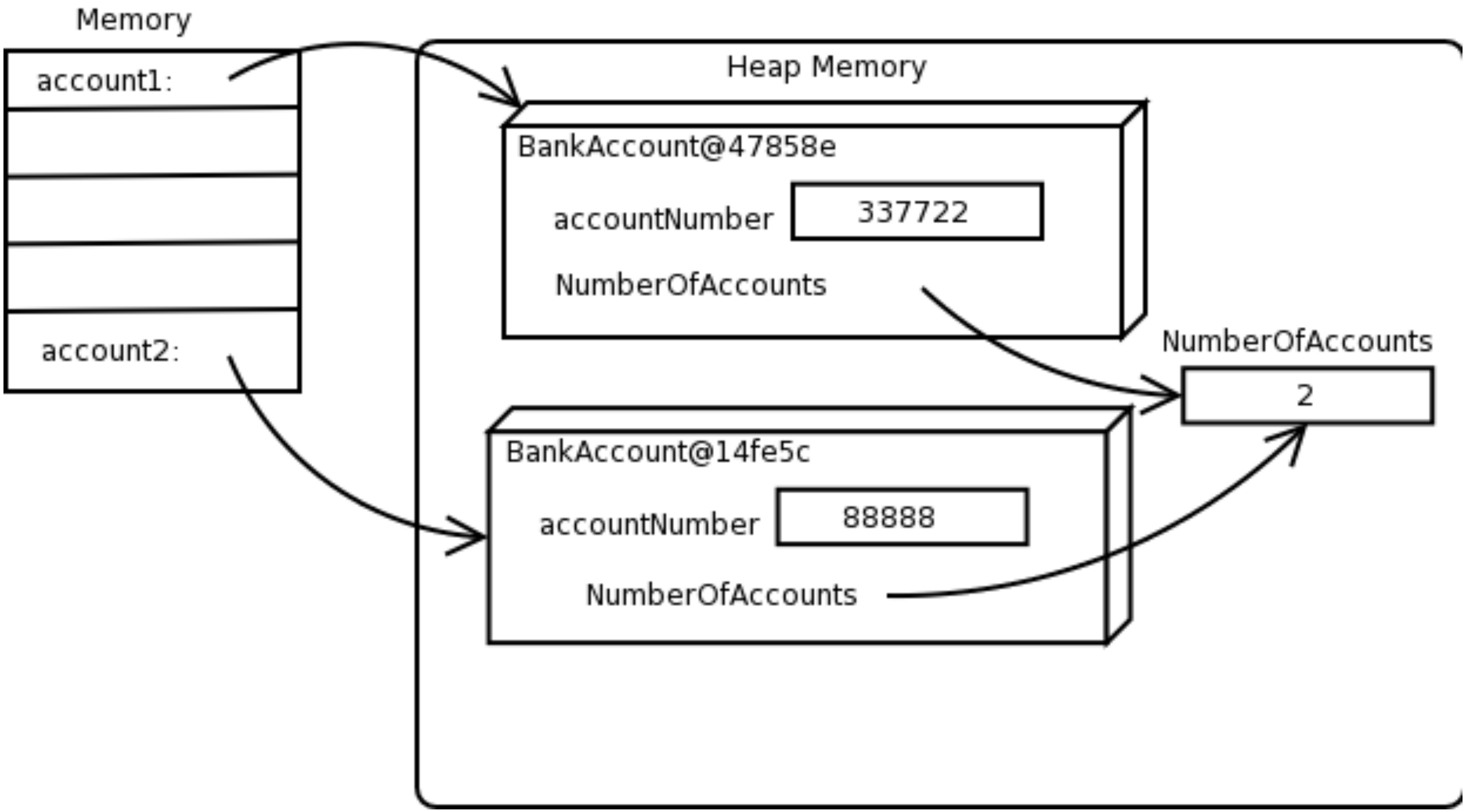


Fig 5-10: **Shared class variable NumberOfAccounts**

Referencing Static Variables

Example 5-21: Static Variables

```
class BankAccount {
  // Class Variables
  static int NumberOfAccounts = 0;
   // Instance Variables
  String accountNumber = null;
  char accountType;
  private float balance;
  int accountStatus;

   // Using static variable in a constructor
  BankAccount(String num, char type, float bal) {
    accountNumber = num;
    accountType = type;
    balance = bal;
    accountStatus = (type == 's')? 100: 0;
    NumberOfAccounts++;
  }
}
```

Static Variable Initialization

- There are three mechanisms in Java to initialize static variables
  - Default initialization – exactly like default instance variable initialization
  - Explicit initialization – exactly like explicit instance variable initialization
  - *Static Initializer* – similar to a constructor
- A static initializer has the same goal as a constructor
  - Initialize the "object"
  - The "object" in this case is the class itself
  - It initializes `static` variables
- A block of code, identified with the keyword `static`, is allowed in the class definition
  - This block of code is executed when the `static` variables actually come into existence
  - The `static` block is intended to be used to initialize the `static` variables, nothing more
  - A `static` initializer is executed after both default and explicit initialization

Static Initialization Block

Example 5-22: Static Variables

```
class BankAccount {
  // Class Variables
  static int NumberOfAccounts;
  // Instance Variables
  String accountNumber = null;
  char accountType;
  private float balance;
  int accountStatus;

  static { //static initializer
    NumberOfAccounts = 0;
  }

  // Using static variable in a constructor
  BankAccount(String num, char type, float bal) {
    accountNumber = num;
    accountType = type;
    balance = bal;
    accountStatus = (type == 's')? 100: 0;
    BankAccount.NumberOfAccounts++;
  }
}
```

Static Methods

- Java allows methods that are associated with the class
  - They are not associated with any specific object
  - You can access the methods without an object
  - All `static` methods exist independently of any objects
- Static methods are typically used for
  - Library functionality (`Math.abs()`, `Integer.parseInt()`)
  - Implementing certain design patterns (*Factory*, *Singleton*, etc)

Static Methods

- There are some rules when dealing with `static` methods
  - You cannot reference instance variables from a `static` method
  - You cannot reference instance methods from a `static` method
  - However, instance methods can access `static` variables and `static` methods
- Static methods are accessed using the dot-notation
  - The reference variable becomes the class name

```
  Math.abs(-1234);
BankAccount.incrementCount();
```

  - It is also possible to use an instance as the reference variable, though it is consider poor programming style

Static Method Example

Example 5-23: Static Methods

```
class BankAccount {
  // Class Variables
  private static int NumberOfAccounts;
  // Instance Variables
  String accountNumber = null;
  char accountType;
  private float balance;
  int accountStatus;
  static {
    NumberOfAccounts = 0;
  }
  // Class Methods
  private static void incrementCount() {
    BankAccount.NumberOfAccounts++;
  }
  public static int numActs() {
    return BankAccount.NumberOfAccounts;
  }
. . .
```

Static Method Example (cont.)

Example 5-23: Static Methods (continued)

```
    // Using static variable in a method
    BankAccount(String num, char type, float bal) {
      accountNumber = num;
      accountType = type;
      balance = bal;
      accountStatus = (type == 's')? 100: 0;
      incrementCount();
    }
}
```

Static Method Example (cont.)

Example 5-24: Invoking Static Methods

```
class Test{
  public static void main(String [] args) {
    BankAccount b = new BankAccount();
    System.out.println("Number of Accounts: "+ b.numActs());
    System.out.println("Number of Accounts: "+
    BankAccount.numActs());
  }
}
```

Final Variables

- Variables can be marked with the `final` keyword
- This indicates that once they are initialized, they cannot be changed
- Typically, there are two types of data that should be `final`
  - Data representing a true constant (like the value of PI)
  - Data that is initialized once and never should be changed again during execution of the application
- Variables can also be `final` and `static`
  - This means they are class wide constants
  - These variables are also typically `public`
- So-called *blank finals* allow a `final` variable to be initialized in a constructor but then never changed again
  - Note that a blank `final` *must* be initialized *only* in a constructor

Blank Finals Example

Example 5-25: Using Blank Finals

```
class BankAccount {
  private static int NumberOfAccounts;
  final String accountNumber;  // blank final
  final char accountType;      // blank final
  private float balance;
  int accountStatus;
  static {
    NumberOfAccounts = 0;
  }
  BankAccount(String num, char type, float bal) {
    accountNumber = num;
    accountType = type;
    balance = bal;
    accountStatus = (type == 's')? 100: 0;
    incrementCount();
  }
  /*--- more class code ---*/
}
```

Comparing Reference Variables

- Reference variables contain a value which describes the location of an object in the heap
- This value is hidden from you
- If you want to compare the value of two references, you can use the equality operator
  - Use the standard == operator for this comparison
  - `if(refVar1 == refVar2)`
  - `if(refVar1 != refVar2)`

Comparing Reference Variables Example

Example 5-26: Reference Variables

```
class Test {
  public static void main(String [] args) {
    String s1 = new String("Hello");
    String s2 = new String("Hello");
    System.out.println
     ("Before assignment: s1 == s2 ->"+ (s1 == s2));
    String s3 = s1;
    System.out.println
     ("After assignment: s1 == s3 ->"+ (s1 == s3));
  }
}
```

Strings and StringBuffers

- Java has two class for dealing with a string of characters
  - `java.lang.String`
  - `java.lang.StringBuffer`
- `String`s can
  - Literals

    ```
    String literal = "String Literal";
    ```
  - Objects

    ```
    String object = new String("String Object");
    ```
- The `String` class provides many methods
  - `length`
  - `substring`
  - `toLowerCase / toUpperCase`
  - `startsWith`
  - Etc.

Strings and StringBuffers

- `String`s, though easy to work with can add overhead on your system
  - `Literals are kept around`
  - `Not garbage collected`
  - `Concatenation can be costly`
- `StringBuffer`s can only be objects

```
StringBuffer object = new StringBuffer("StringBuffer ");
```

- `StringBuffer` does not have operator overloading like `String`

```
object.append("Object");
```

- The `StringBuffer` class provides many methods
  - `length`
  - `substring`
  - `toLowerCase / toUpperCase`
  - `startsWith`
  - Etc.
- `StringBuffer`s do not have the overhead of `String`s

String Object Example

Example 5-27: String class methods

```
class Test {
    public static void main(String [] args) {
        String s1 = new String("Hello");
        String s2 = s1.toUpperCase();
        String s3 = s1.toLowerCase();
        System.out.println("s1 -> "+ s1);
        System.out.println("s2 -> "+ s2);
        System.out.println("s3 -> "+ s3);
    }
```

StringBuffer Example

Example 5-28: StringBuffer used to reverse a String

```
class ReverseString {
    public static String reverseString(String s) {
        // Use a String method to get length of String
        // We need to allocate a StringBuffer of a specific size.
        int size = s.length();
        StringBuffer sb = new StringBuffer(size);

        // Use the charAt() String method to get the character
        // from the String, and then use a StringBuffer method
        // to append it to the StringBuffer.
        for (int index = (size - 1); index >= 0; index--) {
            char c = s.charAt(index);
            sb.append(c);
        }

        // Convert the StringBuffer to a String
        return sb.toString();
    }
  }
```

Arrays as Objects

- Early we discussed arrays as
  - Basic “data structure”
  - With an inherent attributed called `length`
- Arrays in Java are technically objects
- Objects are always created with the new keyword; arrays are no different
- Objects typically contain attributes (instance variables); arrays contain attributes as well, typically called elements
- There is no “constructor” when dealing with an array

Arrays as Objects Example

Example 5-29: Creating arrays

```
class Test {
  public static void main(String [] args) {
    // create the array references
    BankAccount [] a1;
    BankAccount a2 [];
    // create an array
    a1 = new BankAccount[6];
    for (int k = 0; k < 3; k++) {
        a1[k] = new BankAccount();
    }
    // make a1 and a2 point to the same array
    a2 = a1;
}
```

Arrays as Objects

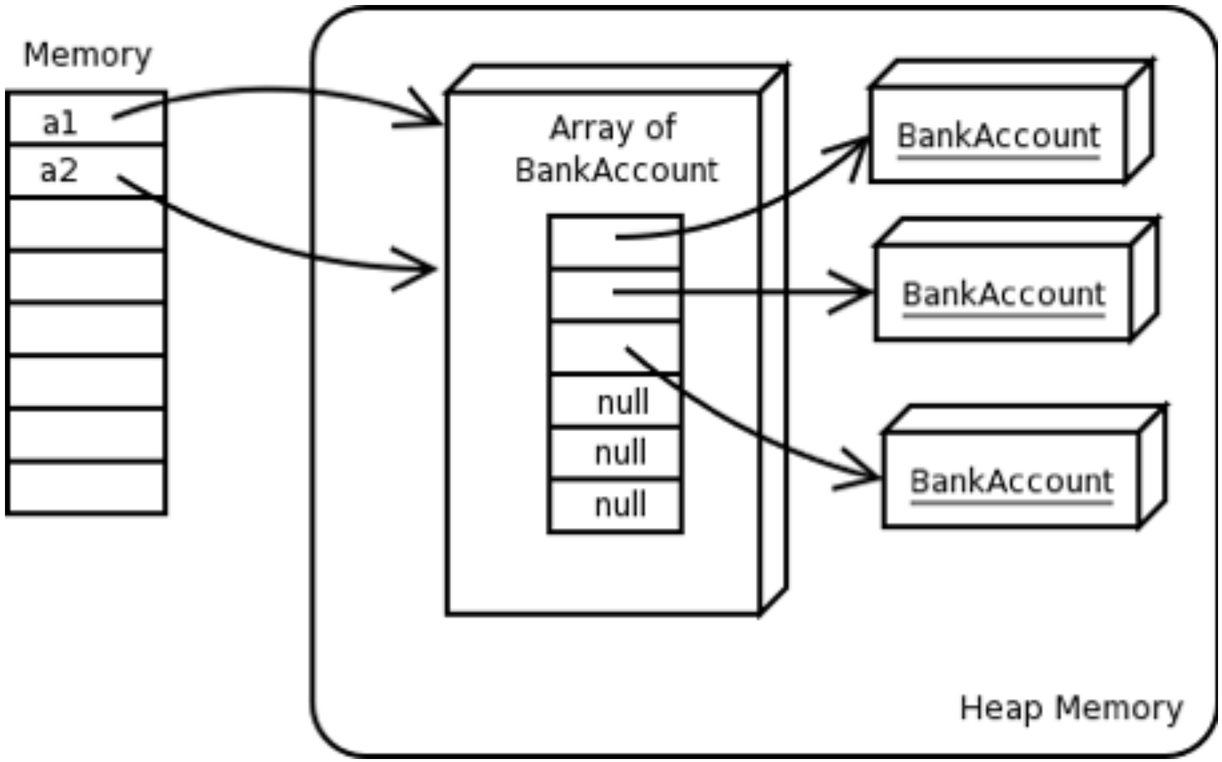


Fig 5-11: **Result of example 5-29**

Copying an Array

Example 5-30: Copying arrays

```
class Test {
  public static void main(String [] args) {
    // create the array references
    BankAccount [] a1;
    BankAccount a2[];
    BankAccount [] a3;
    // create an array
    a1 = new BankAccount[6];
    for (int k = 0; k < 3; k++) {
      a1[k] = new BankAccount();
    }
    // make a2 into a copy of a1 - hard way
    a2 = new BankAccount[a1.length];
    for (int k = 0; k < a1.length; k++) {
      a2[k] = a1[k];
    }
    a3 = new BankAccount[a1.length];
    // make a3 into a copy of a1 -- easy way
    System.arraycopy(a1,0,a3,0,a1.length);
  }
}
```

Copying an Array

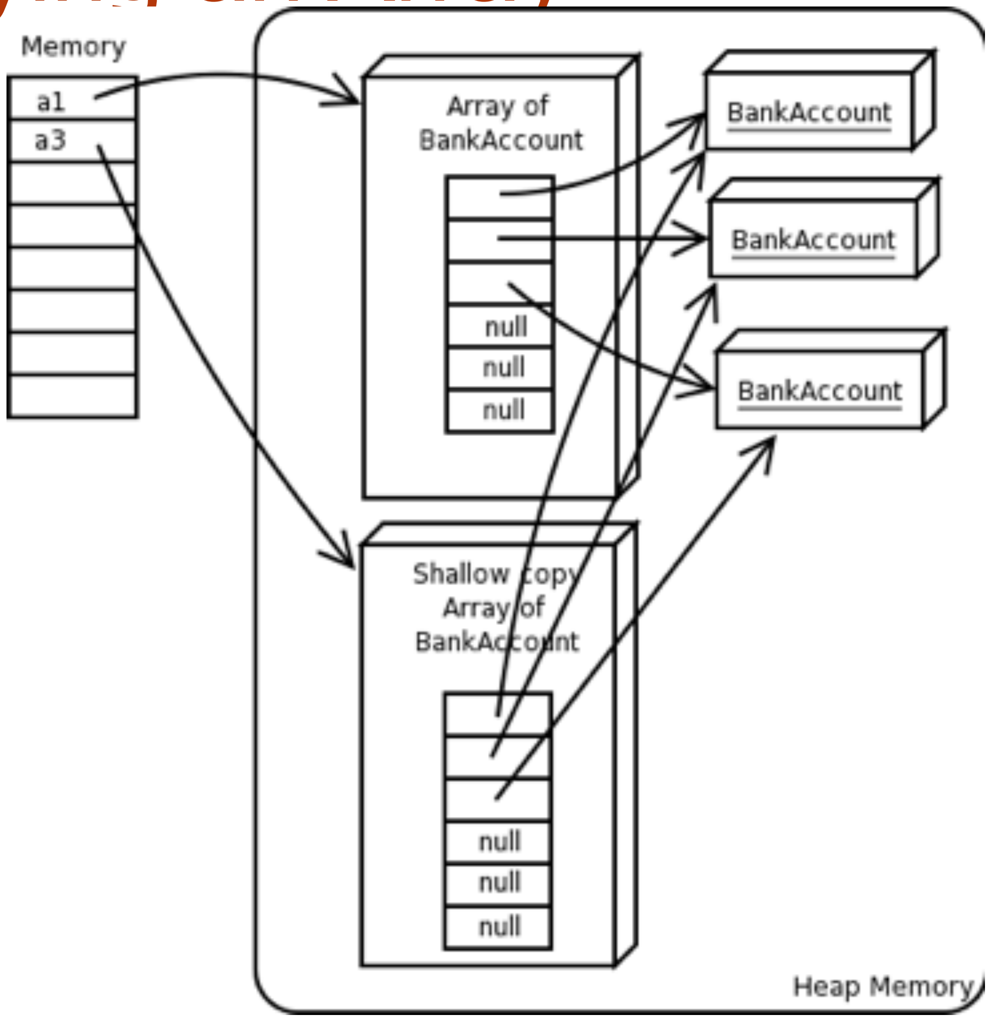


Fig. 5-12: **Result of shallow copy**

Deep Copying Example

Example 5-31: Deep copying arrays

```
BankAccount a1 = new BankAccount[6];
for (int k = 0; k < 3; k++) {
  a1[k] = new BankAccount();
  // make a2 into a deep copy of a1 --
  a2 = new BankAccount[a1.length];
  for (int k = 0; k < a1.length; k++)
  {
    a2[k] = a1[k].clone();
  }
}
```

Deep Copying an Array

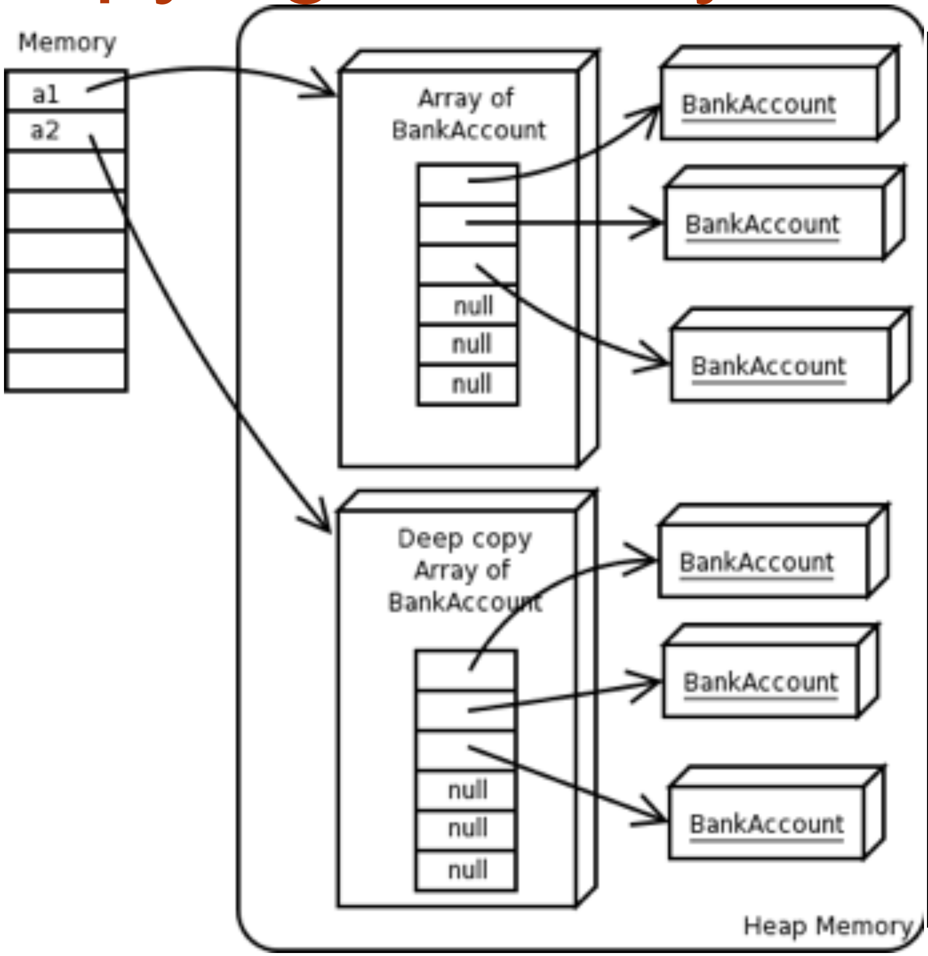


Fig. 5-13: **Result of deep copy**

Wrapper Classes

- Sometimes it is useful to treat primitives as objects
- There is no way to cast a primitive into an object
- Java provides wrapper classes for all primitive types
  - `java.lang.Long`
  - `java.lang.Integer`
  - `java.lang.Short`
  - `java.lang.Boolean`
  - Etc.
- The wrapper classes provide some useful functionality like
  - Converting primitives to and from `String`
  - Retrieving `System` properties as the primitive value
- Even though the wrappers represent primitives, you can not use standard arithmetic operations

Wrapper Classes Example

Example 5-32 Wrapper classes

```
class Test {
  public static void main(String [] args) {
    String s = "7839276";
    long var = Long.parseLong(s);
    System.out.println("Value of var is "+var);
    // Create a Long object to wrap this value
    Long obj = new Long(var);
    // This is an object but we can still get the data
    System.out.println("obj wraps "+ obj.longValue());
  }
}
```

Summary

We covered

- Using `public` and `private` access modifiers
- Class methods and variables
- The `final` keyword with variables
- Using `String` and `StringBuffer` objects
- Using arrays
- Using wrapper classes

Inheritance in Java
(Chpt. 6)

Objectives

At the end of this module, you should be able to:

- Describe the OO concepts of abstraction and inheritance
- Implement inheritance and method over-riding
- Use the `protected` keyword
- Use `abstract` methods and classes
- Use `interface`s
- Use up-casting and down-casting

Abstraction the Real World - Concrete Classes

- Every object is of some type. This is how we naturally think about the world, it allows efficient information processing

```
String s1 = new String();
MyType mt = new MyType();
```

- Types are defined by a process of abstraction or generalization - grouping a collection of objects together based on some common features; and creating a prototype

Abstraction

- Types exist only because the observers define them
- During the design process all types in the problem domain are synthesized down to a set of design classes that are implemented in code
- No "right" collection of classes exist
  - The collection must address the business problem
  - The collection of classes we come up with is just as valid as anyone else's collection
  - We can introduce classes into our application solely for design or implementation reasons
  - Some designs may be more valid than others; your designs will get more elegant over time

Inheritance in the Real World

Let’s explore abstraction by looking at our banking example

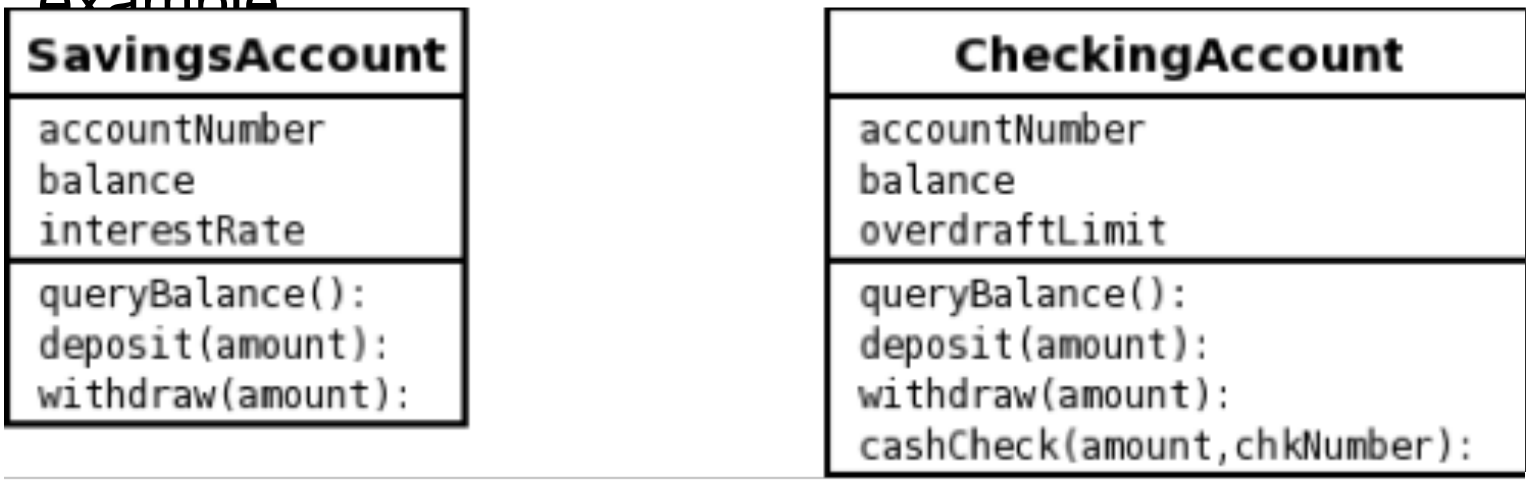


Fig. 6-1: **The two bank account types**

When doing abstraction analysis, look for
- common attributes
- common behaviors

Abstraction and Inheritance

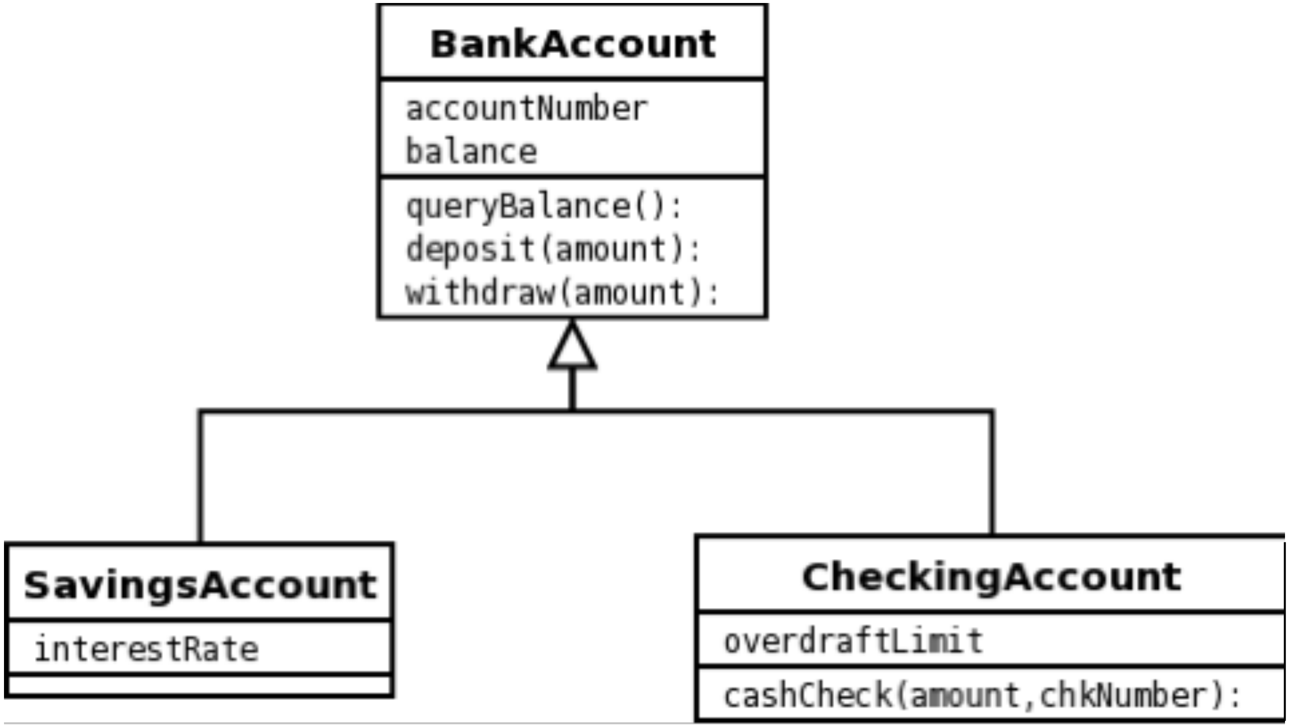


Fig 6-2: **Abstract BankAccount**

Notice where the commonality has been placed, in another class!

Abstraction and Inheritance (cont.)

- "What is the relationship between personal customers and checking accounts?”
- Reason by inheritance - Customers have accounts therefore, personal customers can have accounts
- Checking account is a kind of account, it inherits the property of "being held" by a customer

Abstraction and Inheritance (cont.)

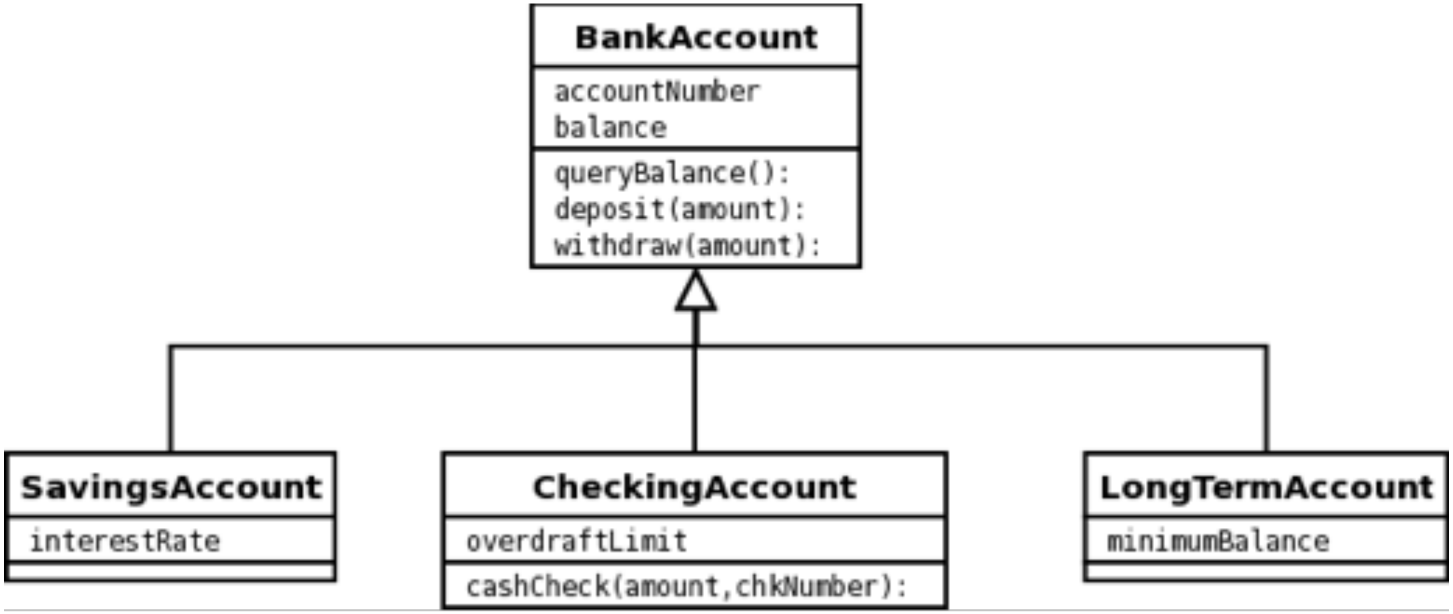


Fig. 6-3: **Adding a LongTermAccount**

Some Jargon

- The `BankAccount` class is called
  - The *base class* or
  - The *super class* or
  - The *parent class*
  - The *generalization*
- The `SavingsAccount` and `CheckingAccount` classes are called
  - *Derived classes* or
  - *Subclasses* or
  - *Child class* or
  - *Specializations*
- Different people have different terms which can be somewhat confusing

Inheritance in Java

- There are two types of inheritance in OOP
  - Multiple inheritance
    - A class can have more than one parent
    - The class inherits traits from both parents; this can cause problems
  - Single inheritance
    - A class can have only one parent
    - Follows along the lines of scientific classification
- Java only supports the single inheritance paradigm
  - This is very different from C++ and C#
  - Easier to manage
  - Less error prone
- Inheritance in Java uses the `extends` keyword

```
class Child extends Parent
```

Inheritance in Java (cont.)

- Inheritance in Java refers to
  - Instance variables
  - Instance methods
- Constructors are not inherited
- You can have control over what variables and methods are inherited using access modifiers
  - `private` variables and methods are not inherited
  - `public`, `protected`, and *default* are, at some level
- `static` variables and methods are not inherited

Inheritance Example

Example 6-1: Implementing inheritance

```
// Superclass BankAccount
class BankAccount {
  float balance;
  String accountNumber;
  float queryBalance() {/*  code */
  float deposit(float amount) { /* code */}
  float withdraw(float amount) {/* code */}
}
// Subclass SavingsAccount
class SavingsAccount extends BankAccount {
  float interestRate;
}
// Subclass CheckingAccount
class CheckingAccount extends BankAccount {
  float overdraftLimit;
  float cashCheck(float amount, int ChkNumber) {
    /* code */
  }
}
```

Final Classes

- Declare a class to be `final` using the `final` keyword preceding the class definition.
- This prohibits the class from being used as the base class in any inheritance structure
- The reasons for declaring a class `final` is always because of a design issue in the application
- For example, we may declare a class `final` because having subclasses would allow objects to circumvent business rules or security checks

Final Class Example

Example 6-2: Final class

```
// Superclass BankAccount is now final
// This will NOT compile
final class BankAccount {
  float balance;
  String accountNumber;
  float queryBalance() {/*  code */
  float deposit(float amount) { /* code */}
  float withdraw(float amount) {/* code */}
}
// Subclass SavingsAccount
class SavingsAccount extends BankAccount {
  float interestRate;
}
// Subclass CheckingAccount
class CheckingAccount extends BankAccount {
  float overdraftLimit;
  float cashCheck(float amount, int ChkNumber) {
    /* code */
  }
}
```

Private Variables Inheritance Example

Example 6-3: Private variables in inheritance

```
// balance is private
// This will NOT compile
class BankAccount {
  private float balance;
  /* -- more code -- */
}
// Subclass SavingsAccount accesses private variable
// balance in super class
class SavingsAccount extends BankAccount {
  public queryBalance() {
    return balance;
  }
}
```

Protected Access Modifier

- Use of the `protected` keyword for instance variables may not be desirable
  - `protected` variables and methods are inherited
  - `protected` variables and methods can also be accessed by other objects of other types
- For strict design correctness, the `private` keyword suffices
- If you need more "protection" than `protected` but not as restrictive as `private` , consider using the default access modifier
  - Inherited by subclasses in the same package
  - Accessed by classes in the same package

Protected Access Modifier Inheritance Example

Example 6-4: Protected variables in inheritance

```
// balance is protected – okay now
class BankAccount {
  protected float balance;
    /* -- more code -- */
}
// Sub class SavingsAccount can access
// protected variable balance in super class
class SavingsAccount extends BankAccount {
  public queryBalance() {
     return balance;
  }
}
```

Implementing Inheritance

Sometimes it is useful for a child to change an inherited behavior

- This can be performed using method over-riding
- Instead of the inherited method being invoked, the over-ridden method will be invoked

Implementing Inheritance Example

Example 6-6: Implementing inheritance

```
class Parent {
  private String priVar = "(Parent Private)";
  protected String proVar = "(Parent Protected)";
  public  String pubVar = "(Parent Public)";
  public void meth() {
    System.out.println("(Parent method)");
    System.out.println(priVar+"  "+pubVar+" "+proVar);
  }
  public void methPar() {
    System.out.println("Parent method");
    System.out.println(priVar+"  "+pubVar+" "+proVar);
  }
}
class Child extends Parent {
  public  String  pubVar = "(Child Public)";
  private String  priVar = "(Child Private)";
  public void meth() { //over-ridden method
    System.out.println("Child method");
    System.out.println(priVar+"  "+pubVar+" "+proVar);
  }
. . .
```

Implementing Inheritence Example (cont.)

Example 6-6: Implementing inheritance (continued)

```
    public static void main(String [] args) {
        Child c = new Child();
        c.meth();
        c.methPar();
    }
    /* Output is
     * Child method
     *(Child Private)  (Child Public) (Parent Protected)
     * Parent method
     *(Parent Private)  (Parent Public) (Parent Protected)
     */
}
```

More About Implementing Inheritance

- Sometimes it is useful for a child to interact with the parent
  - This can be performed using a built-in reference – `super`
  - Use the dot-notation with `super` just like any other reference
- Typically super is used to access an over-ridden method in the parent class
- It can also be used to explicitly access variables in the parent class; however, this may not be required
- In some cases, the parent class will encapsulate sensitive data and behaviors
  - This may prevent method over-riding
  - You may need to use something referred to as *shadowing*
  - Shadowing basically replicates parent encapsulated data and behaviors in the child

More About Implementing Inheritance (cont.)

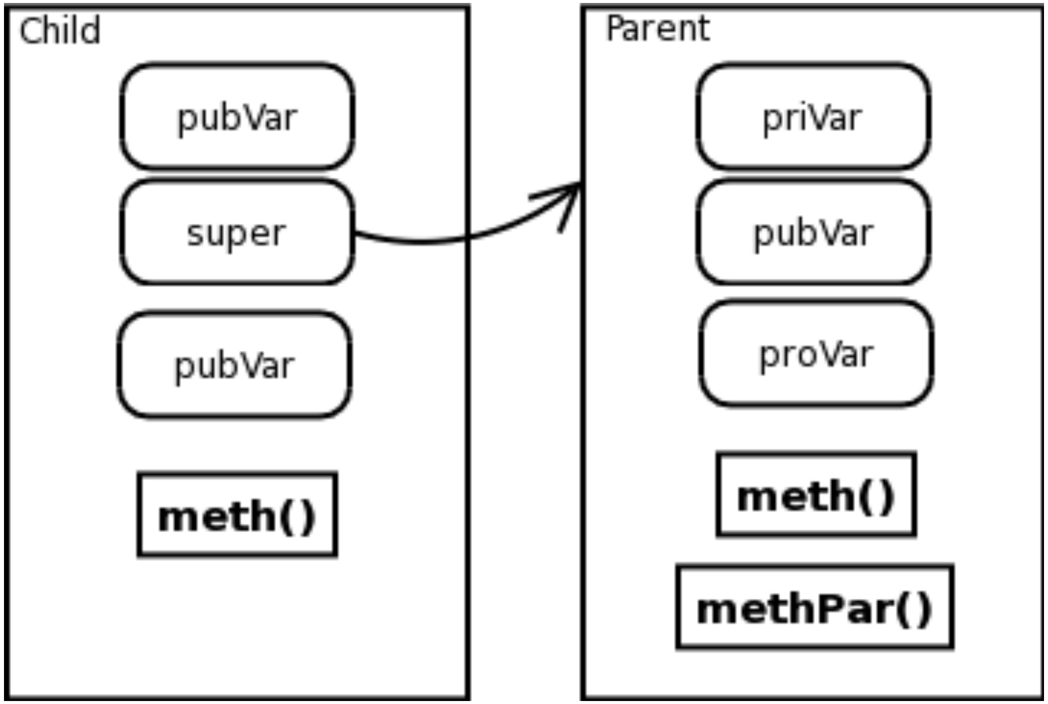


Fig. 6-4: **Result of inheritance from example 6-6**

Shadowing Example

Example 6-7: Implementing inheritance with shadowing

```
class Parent {
  /* just like example 6-6
}
class Child extends Parent {
  public  String  pubVar ="(Child Public)";
  private String  priVar = "(Child Private)";
  public void meth() {
    System.out.println("Child method");
    System.out.println(priVar+ "  " +
                       super.pubVar + " " + proVar);
  }
  public static void main(String [] args) {
    Child c = new Child();
    c.meth();
    c.methPar();
  }
  /* Output is
   * Child method
   * Child Private)  (Parent Public) (Parent Protected)
   * Parent method
   * Parent Private)  (Parent Public) (Parent Protected)
   */
 }
```

Invoking A Parent Method Example

Example 6-8: Implementing inheritance with over-riding

```
class Parent {
  /* just like example 6-6
}
class Child extends Parent {
  public  String  pubVar ="(Child Public)";
  private String  priVar = "(Child Private)";
  public void meth() {
    System.out.println("Child method");
    System.out.println(priVar+"  "+super.pubVar+" "+proVar);
  }
  public void up() {
    System.out.println("Up method");
    super.meth();
  }
  public static void main(String [] args) {
    Child c = new Child();
    c.up();
  }
  /* Up method
   * Parent method)
   * Parent Private)  (Parent Public) (Parent Protected)
   */
}
```

Extending and Implementing Methods

When over-riding a method

- The method signature in the child class can have the same access modifier
- The method signature in the child class can be less restrictive

Example 6-9: over-riding and access modifiers

```
class Parent {
  protected void m1() {}
  public void m2() {}
}

class Child extends Parent {
  // this is allowed public is less restrictive
  public void m1() {}
  // not allowed must be public
  protected void m2() {}
}
```

Accessing Super Class

- Objects are initialized through the use of constructors
- As part of the initialization process, the JVM calls the constructor for each super class in the inheritance chain
- By default, the JVM will call the default or no-argument constructor in the super classes when creating the child object
- In some cases, the creation of the child object will require a different constructor be called in the parent class
  - You need to specify which parent constructor to call
  - Use the `super` keyword in a manner similar to the use of the `this` keyword in the constructor
  - The `super` call must be the first line in the constructor

Accessing Parent Constructors Example

Example 6-10: Super class constructors

```
class Parent {
   Parent() {
     System.out.println("Parent()");
   }
   Parent(int i) {
     System.out.println("Parent(int)");
   }
 }
 class Child extends Parent {
   Child () {
     //redundant.. Default constructor automatically called
     super();
     System.out.println("Child()");
   }
   Child (int i) {
     super(i);
     System.out.println("Child(int i)");
   }
   Child (int i, int j) {
     super(i);
     System.out.println("Child(int i, int j)");
   }
```

Accessing Parent Constructors Example (cont.)

Example 6-10: Super class constructors (continued)

```
    public static void main(String [] args) {
      Child c1 = new Child();
      Child c2 = new Child(1);
      Child c3 = new Child(1,2);
    }
} //end of Child class
/* This produces the output
 * Parent()
 * Child()
 * Parent(int)
 * Child(int i)
 * Parent(int)
 * Child(int i, int j)
 */
```

Abstract Classes

- We have already looked at creating generalizations and specializations of types
  - Created concrete base class
  - Created concrete child class
- Another way, and possibly more common way, to create generalizations and specializations of types is with an *abstract* class
  - Provide description of the required functionality for all specializations
  - However, does not provide implementation
  - Leaves implementation to specialization

Abstract Classes (cont.)

- Abstract classes
  - Are partially defined, partially undefined where as concrete classes are fully defined
  - Can not be instantiated
  - Must be extended to become fully defined
- Abstract classes contain
  - Instance variables and methods
  - Class variables and methods
  - Constructors
  - `abstract` methods

Abstract Classes Example

Example 6-11: Abstract class

```
abstract class Teller {
  String deposit(String amt, String act)
    return "Not implemented";
  }
  String withdraw(String amt, String act)
    return "Not implemented";
  }
  String queryBalance(String act)
    return "Not implemented";
  }
}
class HumanTeller extends Teller {
  public static void main(String [] args) {
      // following line is a compiler error
      // Teller t = new Teller();
    HumanTeller ht = new HumanTeller();
  }
}
```

“Multiple” Inheritance

- Java is a single inheritance language
  - A Child class only ever has one direct ancestor
  - Easy to understand and use
  - May limit more complex classifications of types
- Java supports something like multiple inheritance
  - Not true multiple inheritance
  - Allows objects to take on the behaviors of more than the one ancestor
  - The adopted behaviors are outlined in `interface`s
- `interface`s can be thought of as
  - Abstract, abstract classes (really abstract)
  - Most incomplete class you can have in Java

undefined → defined

`interface`
`abstract class`
`class`

Interfaces

- `interface`s are built like other classes
  - Stored in `.java` files
  - Similar structure, different body

```
<access_modifier> interface LedgerAccount {
  //body
}
```

- `interface`s can contain
  - Nothing – referred to as a *marker interfaces* or *tagging interface*
  - `abstract` methods
  - `public static final` variables (constants)
- Abstract methods are
  - Instance methods without a definition – no body `{ }`
  - Can not be class methods

```
public abstract void doSomething();
```

Interfaces (cont.)

- A class can take on the behavior of an `interface` by
  - Implementing the interface
  - Which means defining all of the inherited `abstract` methods
  - Use the keyword `implements`
- Classes can implement as many `interface`s as required

```
class Child implements Type1, Type2, Type3
```

Interface Example

Example 6-15 Using implements

```
interface LedgerAccount {
  abstract String credit(String amt);
  abstract String debit(String amt);
}

class SavingsAccount extends BankAccount implements LedgerAccout {
  //definition for inherited abstract methods
}

class LoanAccount implements LedgerAccount {
  //definition for inherited abstract methods
}
```

Multiple Interface Example

Example 6-16 Using implements for multiple inheritance

```
interface LedgerAccount {
   abstract String credit(String amt);
   abstract String debit(String amt);
}

interface Persistent {
   public abstract read(String url);
   public abstract write(String(url);
}

class SavingsAccount extends BankAccount implements LedgerAccount, Persistent {
   //definition for inherited abstract methods
}

class LoanAccount implements LedgerAccount, Persistent {
   //definition for inherited abstract methods
}
```

Type Polymorphism

- Polymorphism is
  1. The occurrence of different forms in organisms of the same species
  2. Typically called *type polymorphism* – an object can be many different types
- An object has polymorphism if
  - It has one parent, and many “grandparents”
    - All objects extends from `java.lang.Object`
    - So, by default have type polymorphism
  - It implements more than one `interface`

Type Polymorphism Example

Example 6-17 Type polymorphism

```
interface LedgerAccount {
  /* body code */
}

interface Persistent {
  /* body code */
}

class BankAccount {
  /* body code */
}

class SavingsAccount extends BankAccount implements LedgerAccout, Persistent {
  /* body code */
}


  // Elsewhere in code…

  SavingsAccount s1 = new SavingsAccount();

  LedgerAccount s2 = new SavingsAccount();

  BankAccount s3 = s1;
```

Type Casting

- If an object has more than one type, how do you determine its functionality?
  1. Look at the type of the reference variable
  2. Utilize *type casting*
- Type casting is like casting in primitives
  - “convert” one type to another
  - After conversion may loose precision
    - Primitive example
      `int x = (int) 3.145F;`
    - Reference example
      `s1 = (SavingsAccount) s2;`
  - Java handles widening (*up-casting*) automatically
  - Narrowing (*down-casting*) must be performed manually

Type Casting Example

Example 6-17 Type casting

```
interface LedgerAccount {
  /* code */
}
interface Persistent {
  /* code */
}

class SavingsAccount extends BankAccount implements LedgerAccout,
                                                    Persistent {
  /* code */
}

 // Elsewhere in code…
 SavingsAccount s1 = new SavingsAccount();
 BankAccount s3 = s1; // this is Okay
 s1 = s3;   //illegal
 s1 = (SavingsAccount) s3;  // now this is okay
 LedgerAccount s4 = new SavingsAccount(); // okay
 s1 = (SavingsAccount) s4;  // need to cast here too
```

Summary

We covered

- The OO concepts of abstraction and inheritance
- Implementing inheritance and over-riding
- Using the `protected` keyword
- Using `abstract` methods and classes
- Using `interface`s
- Using up-casting and down-casting

Class Design
(Chpt. 7)

Objectives

At the end of this module you should be able to

- Use encapsulation properly
- Use a business interface
- Use get and set methods
- Use delegation

Enforcing Encapsulation

- Any method or variable that is not `private` is part of an interface
- An interface represents the published attributes and behaviors of an object
- Think of a light switch as an object, what is its interface?
  - Turn On
  - Turn Off
- Behind the scenes the light switch does the appropriate work to turn the light on or off
- The details and hard work are completely hidden from you; the details can change while the interface is preserved

Enforcing Encapsulation

- In Java we have three different access modifiers that relate to the interface:
  1. `public`: presented to world without restriction.
  2. `protected`: package interface *and* all of the `protected` variables and methods
  3. *default*: presented to other objects in the same package
- Proper use of access modifiers allows us to hide the details of how our object works
- Interfaces are one common way to hide the details of an object

Enforcing Encapsulation (cont.)

- Interfaces are one common way to hide the details of an object
- An interface will only ever contain public methods
- For each property that we need to expose to the world, we define a business method
- The interface becomes, what is known as, the *business interface*

Business Interface

Example 7-1: Defining a BankAccount Business interface

```
public interface BankAccount  {
  public String deposit(String amt);
  public String queryBalance();
  public String withdraw(String amt):
}

class BankAccountImp implements BankAccount {
    // What used to be our BankAccount class
}
```

Accessors and Mutators

- Typically it is good OO design to use layers of abstraction to decouple interface from implementation
- In our example, `BankAppImpl` could directly interact with the member variables to `queryBalance`
- However, we usually use get and set methods to read and write set the value of variables
- The get and set methods are then used in the business methods
- This provides abstraction between the interface and the implementation

Get And Set Methods

Example 7-2: Get and Set methods

```
public interface BankAccount  {
   public String deposit(String amt);
   public String queryBalanceUS();
   public String queryBalanceCan();
   public String withdraw(String amt):
}

class BankAccountImp implements BankAccount {
  private float balance;
  private float getBalance() { return balance’}
  private boolean setBalance(float amt) {
    if (amt < 0.0 f) return false;
    balance = amt;
    return true;
  }
  public String queryBalanceUS() {
    return "" + getBalance();
  }
  public String queryBalanceCan() {
    return "" + (1.38f * getBalance();
  }
}
```

Cohesion & Coupling

- **Cohesion**
  - Each class specializes in one particular responsibility
  - It is the only class that has that responsibility
- **Coupling**
  - The amount of interconnections between components
  - The lower the coupling, the more robust the system
- Design high cohesion classes
  - Break complex classes into simpler ones
  - `switch` statement or variables that refer to a class type suggest creating a super class and use polymorphism
  - A large number of methods suggest that a class is not cohesive enough
  - A large number of member variables suggest that a class is not cohesive
  - Factor out commonality

Factor Out Common Code

Example 7-3: Constructors

```
class BankAccountImp implements BankAccount {
  private float balance;
  BankAccount(float initbal) throws AccountCreationException {
    if (!setBalance(inialbal)
        throw new AccountCreationException();
  }
  private float getBalance() { return balance’}
  private boolean setBalance(float amt) {
    if (amt < 0.0 f) return false;
    balance = amt;
    return true;
  }
  public String queryBalanceUS() {
    return "" + getBalance();
  }
  public String queryBalanceCan() {
    return "" + (1.38f * getBalance();
  }
}
```

Utility Classes

- Java only allows a method to return a single value
- The value can be either
  - Primitive
  - Reference
- How do we return more than one value?
  - Create a utility class
  - Sometimes known as a *Value Object*
  - Or a *Holder*
- The holder contains the collective data you are trying to pass back
  - Typically the data is read-only
  - Possibly final

Utility Classes

Example 7-4: Return Objects

```
class Receipt {
   boolean succeeded;
   float balanceBefore;
   float balanceAfter;
   String transActionID;
   String message;
   // appropriate methods
}
public interface BankAccount  {
   public Receipt deposit(String amt);
   public Receipt queryBalance();
   public Receipt withdraw(String amt):
}
```

Utility Classes

Example 7-5: Request Objects

```
class Receipt {
  boolean succeeded;
  float balanceBefore;
  float balanceAfter;
  String transActionID;
  String message;
  // appropriate methods
 }
 class Request {
  String Account;
  String Amount;
 }
 class WithdrawRequest extends Request {};
 public interface BankAccount  {
   public Receipt transaction(Request req);
 }
```

Delegation & Inheritance

- Inheritance, in particular, method over-riding has been used to create different behaviors for related types
- However, inheritance has its limitations
  - Have to create many subclasses to cover all the permutations of behavior implementation
  - Can become clumsy if you need one subclass to perform a combination of the permutated behaviors
- Delegation allows us to hand off the responsibility to a third party
- Delegation utilizes separate specific objects for each required behavior
  - Minimizes the code required to cover permutations
  - Provides flexibility by changing delegate

Delegation & Inheritance

Example 7-5: Defining an Oracle

```
class BankOracle {
  public boolean isAccountOK(BankAccount b);
}

class BusinessHoursOracle extends BankOracle {
  public boolean isAccountOK(BankAccount b) {
    /* logic for business hours */
  }
}

class AfterHoursOracle extends BankOracle {
  public boolean isAccountOK(BankAccount b) {
    /* logic for business hours */
  }
}
```

Delegation & Inheritance

Example 7-6: Implementing the Oracle

```
class BankAccount {
  private BankOracle sage;
  BankAccount(BankOracle b) {
    dage = b;
  }
  public BankOracle setOracle(BankOracle newb) {
    BankOracle oldb = sage;
    sage = newb;
    return oldb;
  }

  /* -- later in the class -- */
  public String withdraw(Sting amt) {
    if (!sage.isAccountOK(this))
      return "Failed: Account not available";
        /* --- rest of the code --- */
  }
}
```

Summary

We covered using

- Encapsulation properly
- Business interface
- get and set methods
- Delegation

Java Packages
(Chpt. 8)

Objectives

At the end of this module, you should be able to

- Explain what Java packages do
- Interpret and use fully qualified class names
- Use the `package` statement correctly
- Use the `import` statement
- Understand how *import on demand* works

Packages

- Allow developers to encapsulate collections of related classes and interfaces into larger aggregations
- Do not exist as objects or concrete constructs in the way that classes or interfaces exist
- Exist as *logical groupings* of classes
- Described in a way understood by JVM – *namespace*

Packages (cont.)

- Java SE, J2EE, J2ME are collections of packages
- Java SE provides the core packages for the language
  - `java.lang`
  - `java.net`
  - `java.util`
- J2EE and J2ME provide packages that are extensions to the language
  - `javax.ejb`
  - `javax.servlet`
  - `javax.message`

Java Packages Perspectives

Two perspectives to consider when thinking about packages

- **Design**
  - How to choose packages
  - How to choose classes for packages
  - How to choose package interfaces
- **Implementation**
  - How packages are defined
  - How packaged classes are accessed in code
  - How the compiler and JVM manage and work with packages

Java Package Design

Choosing a package name

- Package names should provide some human-understandable grouping of classes
  - Can have multiple levels separated by periods
  - Each level must be a valid Java identifier
  - Convention uses only ASCII lower case letters
- Package names should be unique to preserve namespace
  - Namespaces are used by the class loading and security mechanisms
  - Namespaces are used in code to refer to classes
- Some package prefixes are reserved
  - `java.`
  - `javax.`
  - `sun.`

Java Package Design (cont.)

- Consider reversing your fully qualified domain name and using it as the prefix
  - `com.  .`
  - `com.apple.`
  - `com.level3.`
- After establishing the base prefix
  - Determine the sub-packages
    - Sub-packages are logical, not physical
    - Types of groupings
    - Ordering of groupings
  - Work from most generic to most specific
    - `com.  .training.java.intro.labs`
    - `com.  .training.java.intro.solutions`
    - `com.  .bankapp`
    - `com.  .bankapp.util`

Defining Java Packages

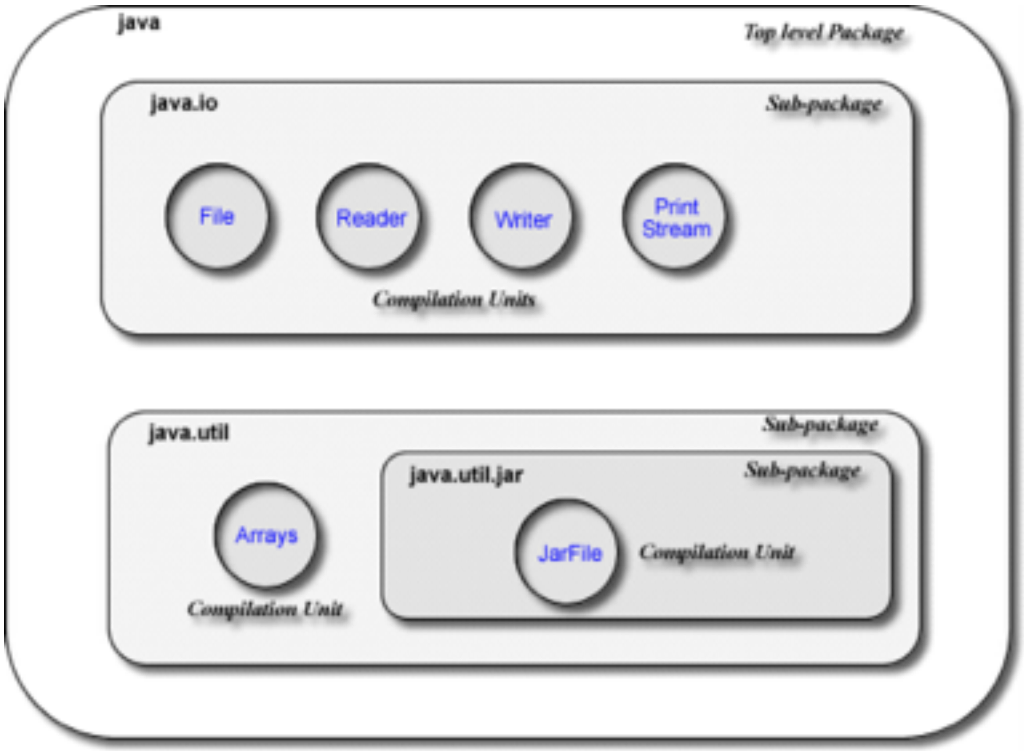


***Fig 8-1*: Java package organization**

Package Implementation

- A class always belongs only one package
  - Explicit package statement
  - Implicit – becomes part of *default package*
- Classes are tied to a package in their source
  - Include a package statement as first executable line in code
  - Can only be one package statement per source file

```
//comments
package com. .blue;
//more comments
class Sky {
  /* body */
}
```

- There is no limit on the number of classes in a package
- The package a class belongs to will effect the use of *default* and `protected` constructs in other classes

Package Implementation Example

Example 8-1: Defining a package

```
package com. .blue;
class Sky {}
class Sea {}
```

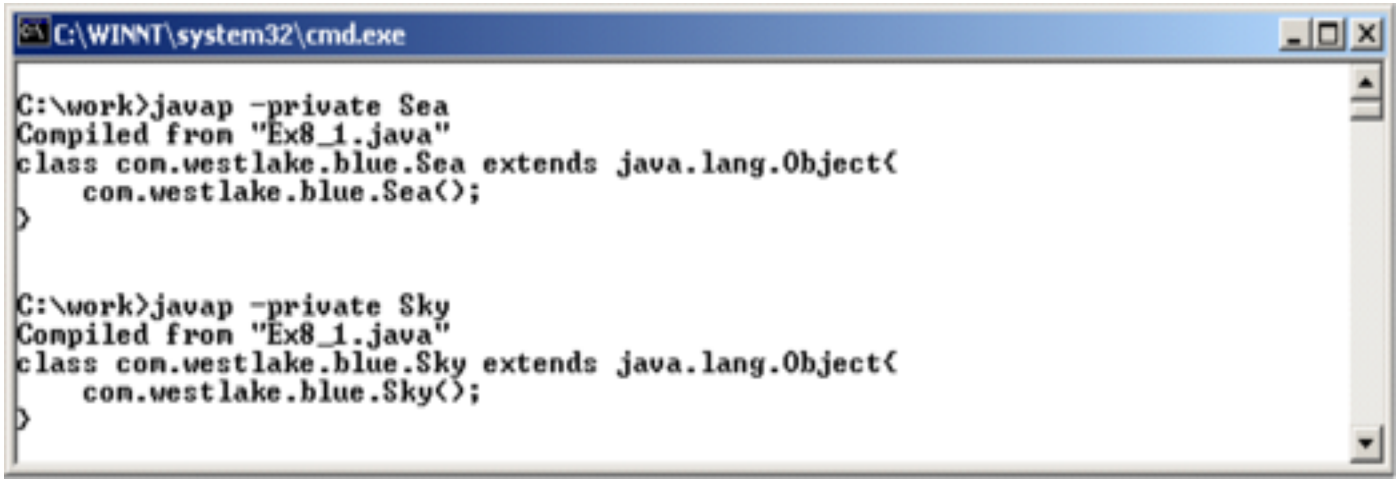


Fig 8-2: **Disassembled contents of the Sky and Sea class files**

Accessing Classes in Packages

There are two scenarios to consider when accessing classes

1. Accessing classes belonging to the same package as the class itself
   - No need to do anything fancy
   - Already have access to classes in the same package
2. Accessing class belonging to a different package than the class itself
   - Access them using *fully-qualified class name* or
   - Access them through the use of *importing*

NOTE: Classes belonging to the `java.lang` package are automatically and always accessible without using fully-qualified class name or importing

Fully Qualified Class Name Example

Example 8-2: Using fully qualified names

```
public class Ex8_2 {
  public static void main(String [] args) {
    java.util.Date d1 = new java.util.Date(8987811L);
    java.sql.Date d2 = new java.sql.Date(8987811L);
    System.out.println("java.util.date is " + d1);
    System.out.println("java.sql.date is " + d2);
  }
}

// Output of the above is
java.util.date is Wed Dec 31 21:29:47 EST 1969
java.sql.date is 1969-12-31
```

Importing Classes

- Using fully qualified class names works
  - Very explicit
  - Easy to read, maintain
  - Laborious to type
- Importing classes is kind of a short cut
  - Use an `import` *statement*
    - Should follow the package statement
    - Typically `import` language packages first
    - Then `import` application specific packages
  - Gives class access to classes in other packages
  - Can `import` as few or as many packages and classes as you need

```
import java.net.Socket;  //access to single class
import java.util.*;  //access to all classes
```

  - Compiler converts all imported class references to fully qualified class name references
  - May experience class name collisions

import Statement Example

Example 8-3: Using the import statement

```
import java.util.Date;
public class Ex8_3 {
  public static void main(String [] args)  {
    Date d1 = new Date(8987811L);
    java.sql.Date d2 = new java.sql.Date(8987811L);
    System.out.println("Java.util.date is " + d1);
    System.out.println("Java.sql.date is " + d2);
  }
}

// Output of the above is
Java.util.date is Wed Dec 31 21:29:47 EST 1969
Java.sql.date is 1969-12-31
```

The import On Demand

Example 8-4: Using import on demand

```
import java.util.*;
public class Ex8_4 {
  public static void main(String [] args)  {
    Date d1 = new Date(8987811L);
    java.sql.Date d2 = new java.sql.Date(8987811L);
    System.out.println("java.util.date is " + d1);
    System.out.println("java.sql.date is " + d2);
  }
}

// Output of the above is
java.util.date is Wed Dec 31 21:29:47 EST 1969
java.sql.date is 1969-12-31
```

Environment Constraints

- Packages map to directory structures
  - The source for classes defined in packages *should* exist in a directory structure that maps to the package name
  - Compile the classes at the top of the containing directory structure
  - Classes will be compiled into the same directory structure, though it may be in a different location
- Compiler and JVM rely on `classpath` values to find classes belonging to packages
  - Can be environment variable
  - Can be passed to compiler and JVM as arguments
- `classpath` should contain a “path” to your classes
  - Could be path to directory structure
  - Could be path to an archive, like a `JAR` or ZIP

Classpath Example

Example 8-5: Packages to Directories

```
// File blue.java in Examples\Mod08\
package blue;
class Sky{}
class Sea{}

// File green.java in Examples\Mod08\
package blue.green;
class Grass {}
class Light {}

// File fruit.java in Examples\Mod08\
package blue.fruit;
class Berry {}
class Plum {}

// File grey.java in Examples\Mod08\
package blue.green.grey;
class Shadow {}
class Sky {}
```

Classpath Example (cont.)

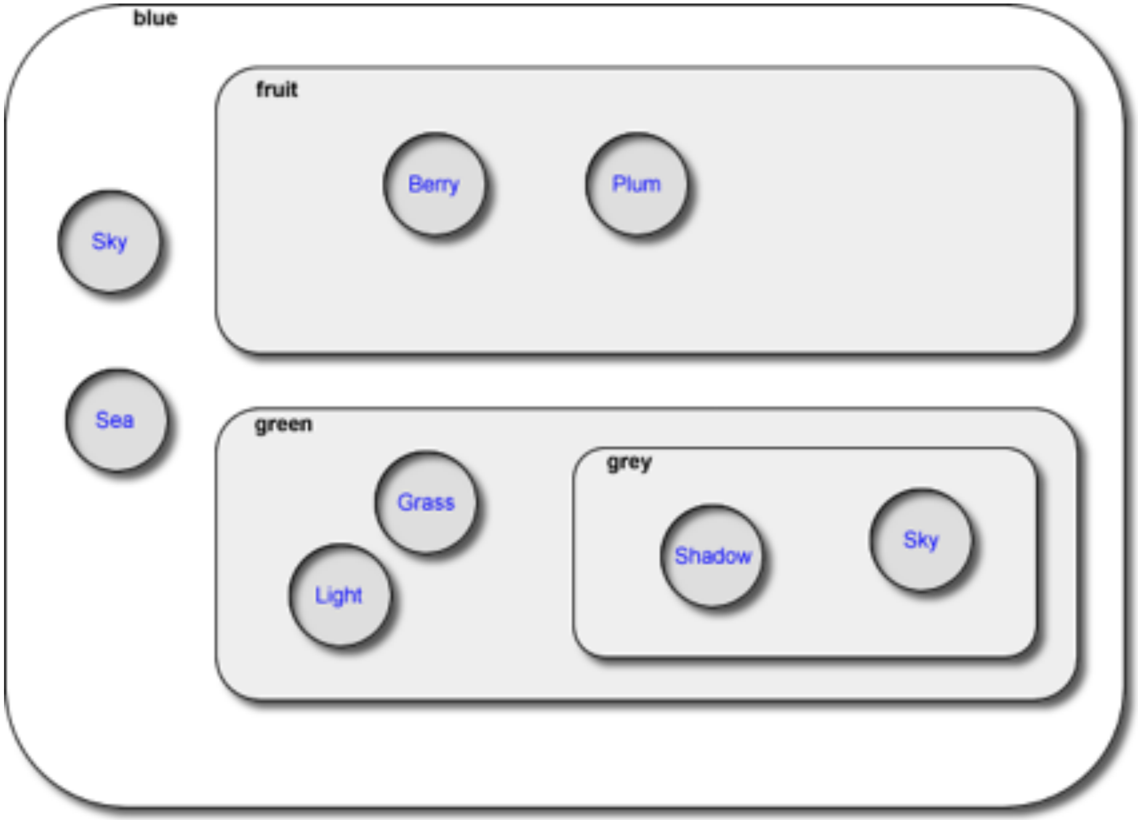


Fig 8-3: **Logical package structure of example 8-5**

Classpath Example (cont.)

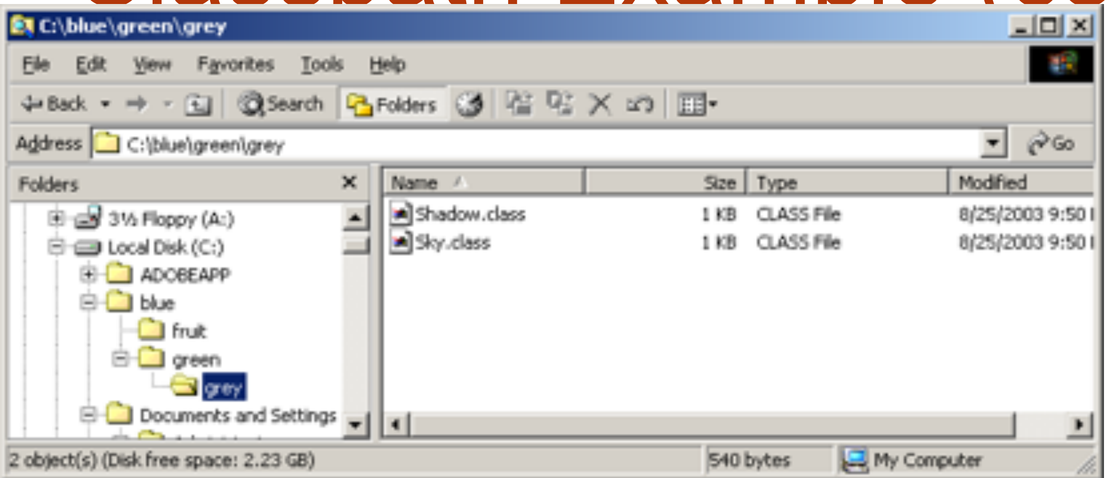


Fig 8-4: **Directory structure of example 8-5**

The `classpath` would have to include a path to a directory or a path a JAR containing the classes in order to find `blue.green.grey.Sky` and `blue.green.grey.Shadow`

Example 8-6: Referencing a jar file

Assume that we have code in the jar file `mycode.jar` and the jarfile is in the directory `c:\mylibs`

```
CLASSPATH=.;c:\;c:\mylibs\mycode.jar
```

Package Facades

- We have already discussed encapsulation at the class and object level
- There is another type of encapsulation, package level
- Commonly referred to as package facades
  - Hides details of the package contents
  - Simplifies business interface of package
  - Provides user flexibility through abstraction

Package Facades

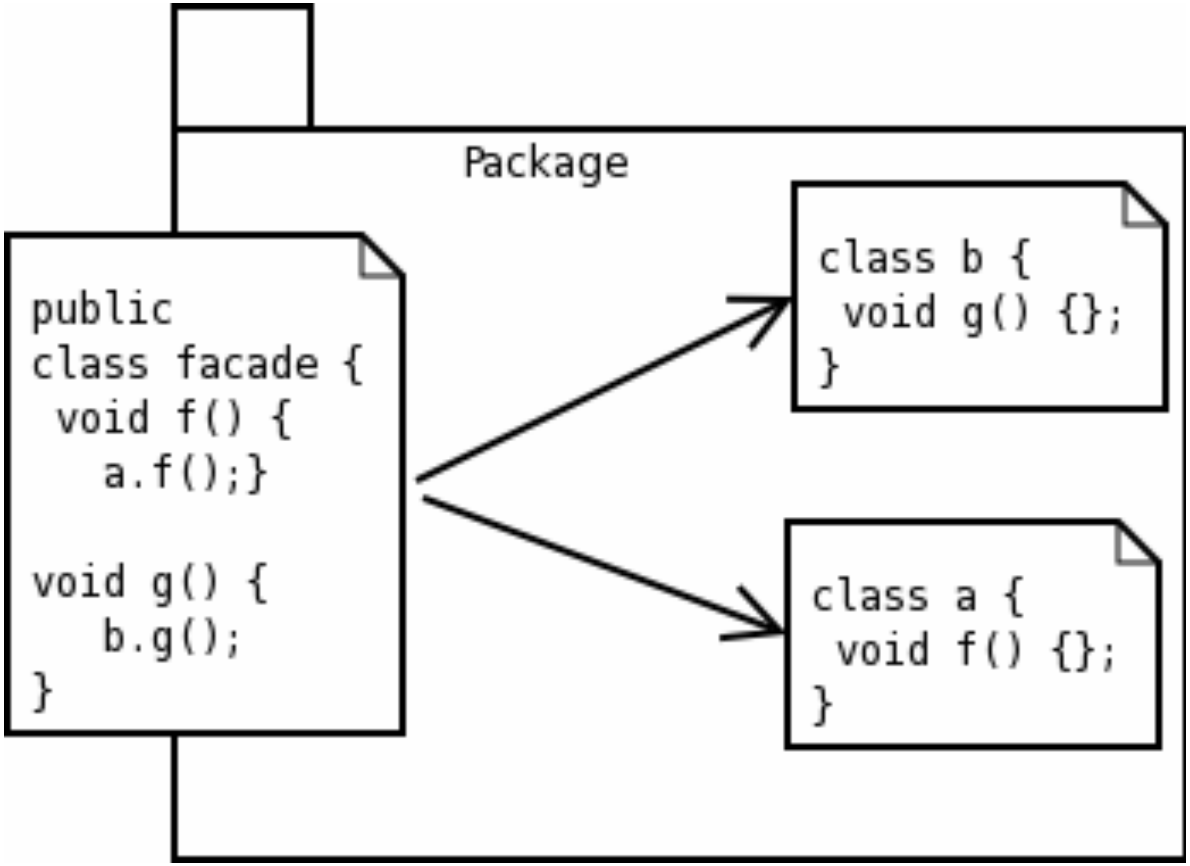


Fig 8-5: **A Façade pattern implementation**

Package Facades

Example 8-7: Implementing a façade

```
//First we have two implementation classes
class imp1 {
  int method1(int x) { return x;}
}
class imp2 {
  void method2() {}
}
//And the Interface defnition for the Façade.
public interface Façade {
  public int method1 (int x);
  public void method2();
}
```

Package Facades (cont.)

Example 8-7: Implementing a façade (continued)

```
// The Implementation of the Façade
public FacadeImp implements Façade {

  private imp1 obj1;
  private imp2 obj2;

  public FacadeImp () {
    obj1 = new imp1();
    obj2 = new imp2();
  }

  public int meth1 (int x) {
    return obj1.meth1(x);
  }

  public void meth2 () {
    obj2.meth2();
  }

}
```

Summary

We covered

- What Java packages do
- Interpreting and using fully qualified class names
- Using `package` statement correctly
- Using `import` statement correctly
- How `import` on demand works

Exceptions
(Chpt. 9)

Objectives

At the end of this module you should be able to

- Describe exceptions & understand their importance
- Describe the Java exception hierarchy
- Define an application exception hierarchy
- Use the `try-throw-catch` construct
- Use nested `try` blocks
- Declare method signatures with `throws`
- Use the `finally` clause
- Show how re-throwing exceptions work

Exceptions

- Java incorporates an exception handling mechanism into the language structure
  - Similar to exception handling in C++ / C#
  - Virtual machine does most of the work to make exceptions work
- Objects that represent something exceptional occurred
  - Could be an exceptional case
  - Could be the expected negative result of a behavior
  - Could be the unexpected negative result of a behavior
  - If handled properly, are recoverable
- Standard Java objects with a specific type hierarcy

Exceptions (cont.)

- Exceptions is not always synonymous with bug
  - Programming faults (bugs)
  - System faults like a down network (not a bug)
- Can manage exceptions - which means either:
  - Code responds to an exception so a problem can be fixed and then continue processing
  - Shutting the application down gracefully in order to do as little damage as possible

Exceptions (cont.)

Exception handling is kind of like event programming

- An interaction generates an event
- The event represents some specific interaction occurred
- The event is sent to the JVM
- The JVM delivers the event to the handler

Exceptions

The basic exception-handling model

- *Try* to perform the interaction

```
try {
  result = getResult();
```

- The interaction fails - an exception occurs

```
  if(result != expectedResult) {
```

- An exception object is created – generate the “event”

```
    BadResultException bre = new BadResultException(“Unexpected Result”);
```

- The exception object is *thrown* – pass the “event” to the JVM

```
    throw bre;
  }
```

- An exception handler *catches* and recovers – JVM delivers “event” to handler

```
} catch(BadResultException bre) {
  //do recovery
}
```

Exceptions

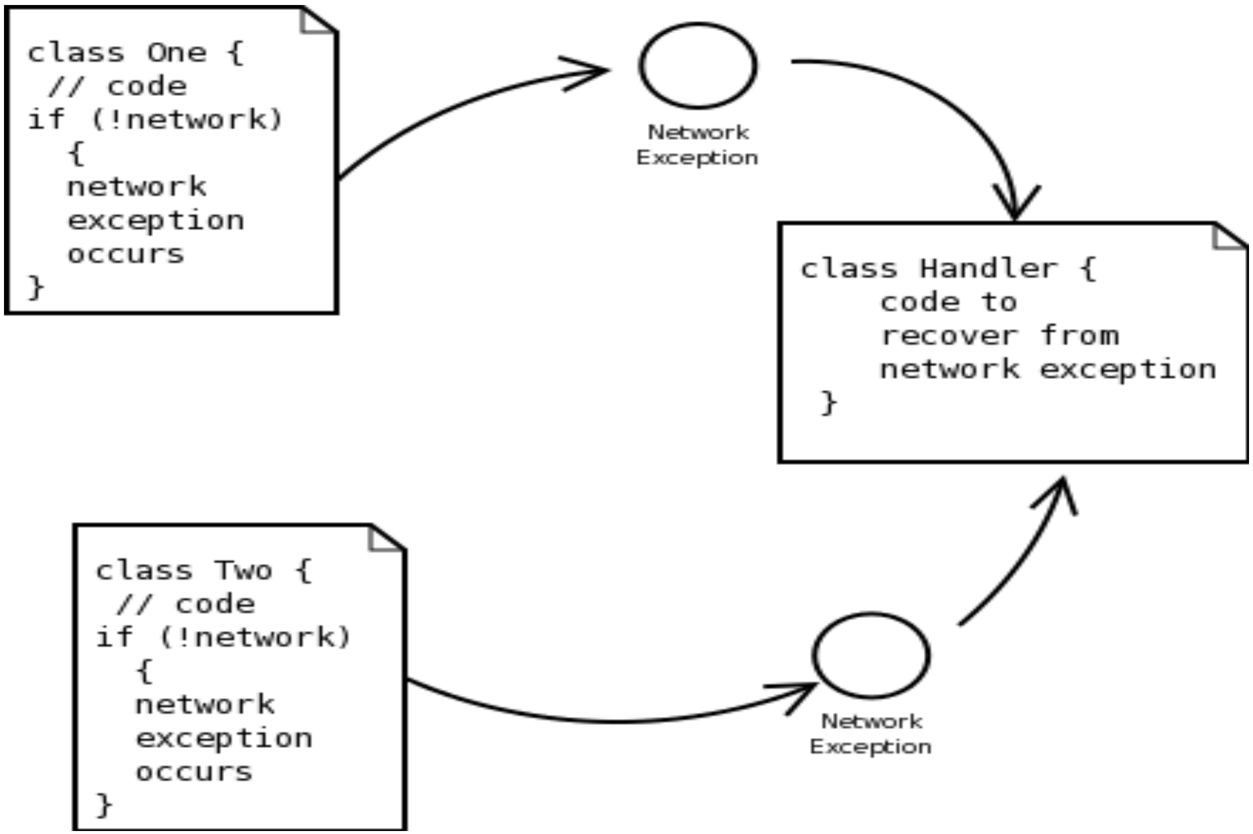


Fig. 7-1: **Throwing an Exception**

Exception Classification

- All exceptions are Java objects
- They are specific types of Java objects
  - `Throwable`
  - Can cause execution flow to be redirected
  - Typically you won't work directly with `Throwable`
- Two types of problems can occur
  - Unrecoverable – `Error`
    - `OutOfMemoryError`
    - `StackOverflowError`
  - Recoverable – `Exception`
    - `NullPointerException`
    - `ArrayIndexOutOfBoundsException`
    - `IOException`

Exception Classification (cont.)

There are two types of Exceptions

- Checked
  - Declared as part of method signature
  - Compiler checks for exception handlers at compile time
  - Typically application-level exceptions
    - `IOException`
    - `SQLException`
- Unchecked
  - `RuntimeException`
  - Not validated at compile time
  - Typically programming bugs
    - `NullPointerException`
    - `ArrayIndexOutOfBoundsException`

Classification of Exceptions in Java

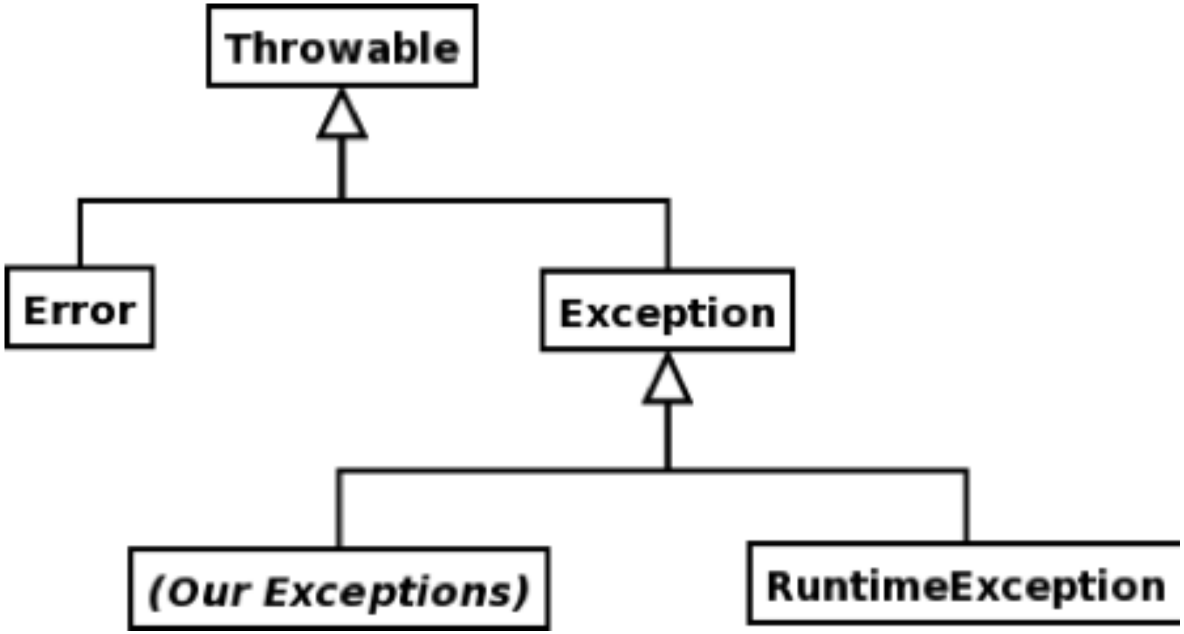


Fig. 7-2: Java Exception Hierarchy

Exception Handling Constructs

Exception handling relies on three main constructs

- `try`
  - Contains code that may fail
  - Flow control moves from `try` to `catch` when exception occurs
- `catch`
  - Contains the detection mechanism
  - Contains the recovery mechanism
  - Executed only if a detected exception occurs
  - Can have multiple `catch` blocks per `try`
- `finally`
  - Used for final clean up
  - Always executed
  - Have one finally block per `try`

```
try {
  //delicate code
} catch (ExceptionType e) {
  //recovery
} finally {
  //final clean up
}
```

Exception Handling Constructs (cont.)

- It is possible to have multiple catch blocks
- When designing multiple exception handlers consider
  - Type of exception
  - Type hierarchy of exception
- These will govern the order of the catch blocks

Specific

Generic

```
try {
   //some network code
} catch(ConnectException ce) {
   //do some connection recovery
} catch (IOException ioe) {
   //do some IO recovery
} catch (Exception e) {
   //do some generic recovery
} finally {
   //do clean up
}
```

Try-Throw-Catch Example

Example 9-1: Exceptions

```
public class Ex9_1 {
   public static void main(String[] args) {
     Ex9_1 testObj = new Ex9_1();
     testObj.exec(args[0]);
   }
   public void exec(String option) {
     try {
       if (option.equals("fail")) {
         throw new Exception();
       }
       if (option.equals("access")) {
         throw new IllegalAccessException();
       }
       System.out.println("No Exception Thrown");
     } catch (IllegalAccessException e) {
       System.err.println("IOExcepton caught");
     } catch (Exception e) {
       System.err.println("Exception caught");
     }
   }
 }
```

Custom Exceptions

- In many cases you will want to create application specific exceptions
- There are two ways
  - Implement the `Throwable` interface
    - Exception become most generic type
    - Much flexibility
    - Have to code all of the details
  - Extend the `Exception` class
    - Build on an existing type
    - Inherit functionality
    - Quick

Implementing an Exception Hierarchy

Example 9-2: Matching base class exceptions

```
class BankException extends Exception {}
class ATMException extends BankException {}

public class Ex9_2 {
  public static void main(String[] args) {
    // here is the try block
    try {
      throw new ATMException();
    } catch (ATMException e) {
      System.err.println("Caught ATMException");
    } catch (BankException e) {
      System.err.println("Caught BankException");
    }
  }
} // Output is Caught ATMException
```

Implementing an Exception Hierarchy

Example 9-3: Matching base class exceptions

```
class BankException extends Exception {}
class ATMException extends BankException {}

public class Ex9_3
  public static void main(String [] args) {
    try {
      throw new ATMException ();
    } catch(BankException e) {
      System.err.println("Caught BankException");
    }
  }
}
// Output is Caught BankException
```

Implementing an Exception Hierarchy

Example 9-4: Matching base class exceptions -- illegal

```
class BankException extends Exception {}
class ATMException extends BankException {}

public class Ex9_4 {
  public static void main(String[] args) {
    // here is the try block
    try {
      throw new ATMException();
     } catch (BankException e) {
      System.err.println("Caught BankException");
     } catch (ATMException e) {
      System.err.println("Caught ATMException");
     }
   }
 }
  // This code will not compile.
  // catch(ATMException e) would never be reached
```

Exception API

- Functionality of Exception is all inherited from Throwable
- Interesting `java.lang.Throwable` APIs
  - `getMessage`
  - `getStackTrace`
  - `initCause`
  - `printStackTrace`
  - `toString`

Implementing an Exception Hierarchy

Example 9-5: Using exception messages

```
class BankException extends Exception {
   BankException(String msg) {
     super(msg);
   }
 }
 public class Ex9_5 {
   public static void main(String [] args) {
       // here is the try block
     try {
       throw new BankException ("I'm a BankException");
     } catch(BankException e) {
       System.err.println(e.getMessage());
       e.printStackTrace();
     }
   }
 }// Output is
  // I'm a BankException
  // BankException: I'm a BankException
  // at Ex9_5.main(Ex9_5.java:19)
```

Nesting

- Java's exception mechanism supports nesting
- You can have
  - Try-catch blocks in `try` blocks
  - Try-catch blocks in `catch` blocks
  - Try-catch blocks in `finally` blocks

Nested Try Blocks

Example 9-6. Nested Try Blocks

```
// Define a couple of exception types
class e1  extends Exception{}
class e2 extends Exception{}

public class Ex9_6 {
  public static void main(String[] args) {
   Ex9_6 testObj = new Ex9_6();
    testObj.exec(args[0]);
  }
  public void exec(String option) {
    // here is the outer try block
    try {
      if (option.equals("outer"))
        throw new e1();
```

. . .

Nested Try Blocks

Example 9-6. Nested Try Blocks

```
            // inner try block
            try {
                   if (option.equals("1"))
                     throw new e1();
                   if (option.equals("2"))
                     throw new e2();
                   System.out.println("No Inner Exception
 Thrown");
            } catch (e2 e) {
              System.err.println("inner e2 caught");
            }
            System.out.println("No Outer Exception Thrown");
      } catch (e1 e) {
        System.err.println("outer e1 caught");
      } catch (Exception e) {
        System.err.println("outer e2 caught");
      }
 }
```

Nested Try Blocks

C:\WINNT\System32\cmd.exe

```
C:\Work>java Ex9_6 pass
No Inner Exception Thrown
No Outer Exception Thrown

C:\Work>java Ex9_6 outer
outer e1 caught

C:\Work>java Ex9_6 2
inner e2 caught
No Outer Exception Thrown

C:\Work>java Ex9_6 1
outer e1 caught

C:\Work>
```

Fig. 9-3 **Output from example 9-6**

Declaring Checked Exceptions

- Checked exceptions are “declared” as part of a method signature
  - Utilizes the `throws` keyword
  - `throws` keyword follows parameter list

```
public void f(String option) throws e1
```

  - The throws clause can specify multiple exceptions

```
public void f(String option) throws e1, e2, IOException, . . .
```

- Declaring exceptions gives compiler and developer ability to recognize an exception may occur and provide the appropriate handling mechanism

Declaring Checked Exceptions Example

Example 9-8. Nested Try Blocks

```
// Define a couple of exception types
class e1  extends Exception{}
class e2 extends Exception{}

public class Ex9_8 {
  public static void main(String[] args) {
     Ex9_8 testObj = new Ex9_8();
    testObj.exec(args[0]); }
  public void g() throws e2 {
    throw new e2(); }
  public void f(String option) throws e1 {
    try {
      if (option.equals("1"))
        throw new e1();
      if (option.equals("2"))
        g();
      System.out.println("No Inner Exception Thrown");
     } catch (e2 e) {
      System.err.println("inner e2 caught"); }
  }
}
```

The Finally Block Example (cont.)

Example 9-9. Finally block

```
// Define a couple of exception types
class e1  extends Exception{}
class e2 extends Exception{}

public class Ex9_9 {
   public static void main(String[] args) {
      Ex9_9 testObj = new Ex9_9();
      testObj.exec(args[0]);
   }
   public void exec(String option) {
      // here is the try outer block
     try {
       if (option.equals("outer"))
          throw new e1();
```

. . .

The Finally Block Example (cont.)

Example 9-9. Finally block

```
            // inner try block
            try {
              if (option.equals("1"))
                throw new e1();
              if (option.equals("2"))
                throw new e2();
              System.out.println("No Inner Exception Thrown");
            } catch (e2 e) {
              System.err.println("inner e2 caught");
            } finally {
              System.err.println("inner finally");
            }
            System.out.println("No Outer Exception Thrown");
        } catch (e1 e) {
          System.err.println("outer e1 caught");
        } catch (Exception e) {
          System.err.println("outer e2 caught");
      }
  }
```

Summary

We covered

- Describing exceptions are & why they are unavoidable as a rule
- Describing the Java exception hierarchy
- Defining an application exception hierarchy
- Using the `try-throw-catch` construct
- Using nested try blocks
- Declaring method signatures with `throws`
- Using the `finally` clause

Java IO
(Chpt. 10)

Objectives

At the end of this module you should be able to

- Describe the architecture of the `java.io` API
- Describe the streams model
- Use implementation streams
- Use filter streams
- Describe the difference between streams, readers and writers
- Use data streams and files
- Use buffered I/O and the `PrintWriter`
- Describe the `File` class

I/O in Java

- All programming languages have I/O facilities
- Java is no different
- Java has two main types of I/O
  - Blocking I/O
    - `java.io`
    - Referred to as synchronous I/O
    - Utilizes streams
  - Non-blocking I/O
    - `java.nio`
    - Referred to as asynchronous I/O
    - Utilizes channels
    - We will not cover NIO

I/O in Java (cont.)

- Blocking I/O has facilities for two types of streams
  - Binary streams
  - Character streams
- A stream is a reference to a “flowing” sequence of bytes
- Anything can generate a stream
  - Network connection
  - Database connection
  - File connection
  - Even `String`

Architecture of a Stream Approach to I/O

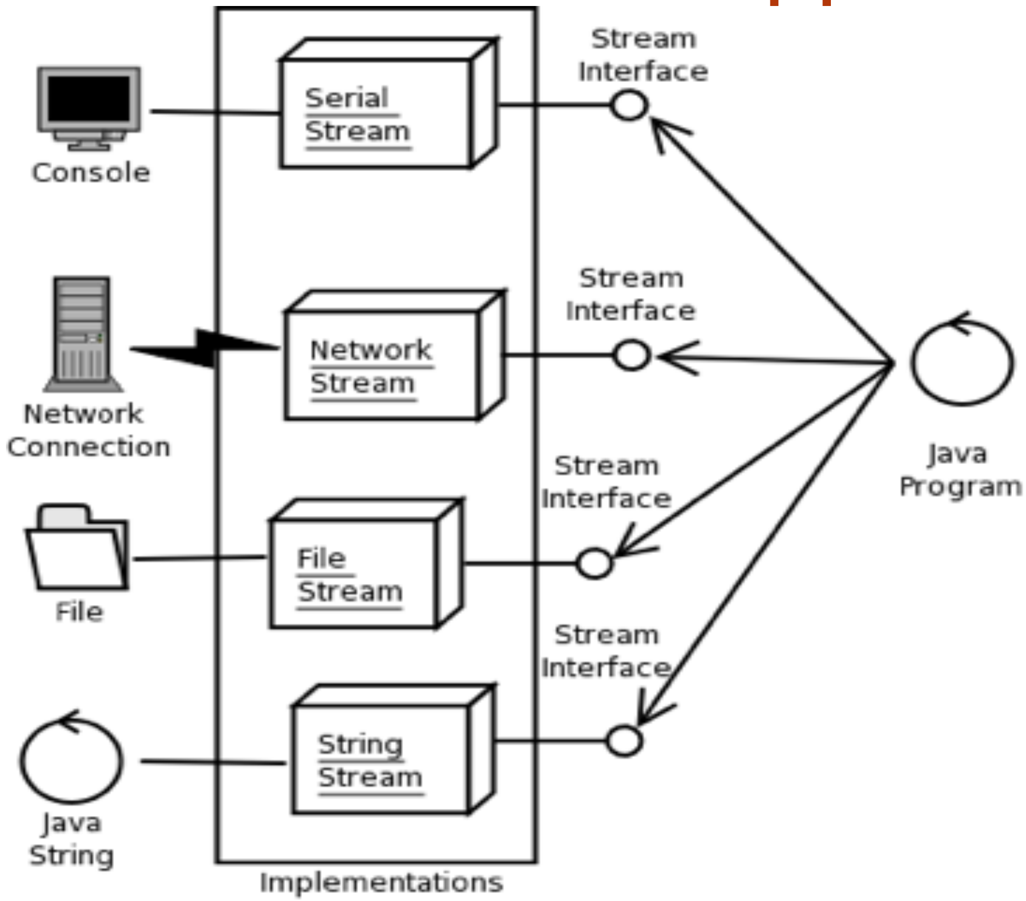


Fig. 10-1: **Logical architecture of a Stream approach to I/O**

Streams Model

The basic stream programming model is the same for binary and character st

1. Open the Stream
    1. Create the stream object
    2. Initialize the stream object
2. Perform operations
    1. Read the data
    2. Write the data
3. Close the stream

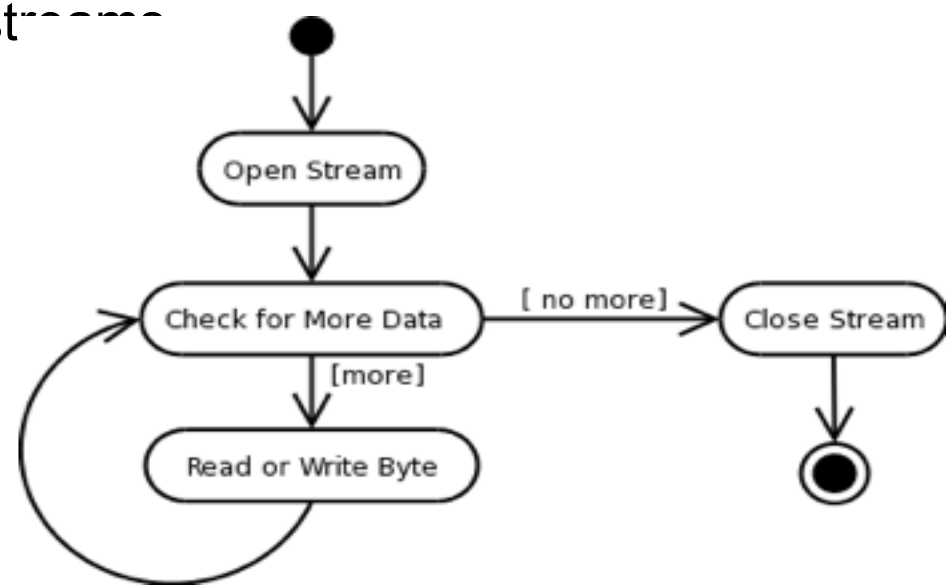


Fig. 10-2: **Basic logic for using an InputStream or OutputStream**

The Top Level interfaces

`java.io.Reader` **Interface**

```
int read()
int read(char cbuf[])
int read(char cbuf[], int offset, int length)
```

`java.io.InputStream` **Interface**

```
int read()
int read(byte cbuf[])
int read(byte cbuf[], int offset, int length)
```

`java.io.Writer` **Interface**

```
int write(int c)
int write(char cbuf[])
int write(char cbuf[], int offset, int length)
```

`java.io.OutputStream` **Interface**

```
int write(int c)
int write(byte cbuf[])
int write(byte cbuf[], int offset, int length)
```

I/O APIs

- The top-level I/O APIs are pretty low-level
  - Good for low-level OS communication
  - Useful when dealing with proprietary protocols
- Java provides higher-level I/O APIs
  - Rely on low-level I/O APIs
  - Provide convenience input and output methods
  - Many variations
    - `FileReader / FileWriter`
    - `FileInputStream / FileOutputStream`
    - `PrintWriter / PrintStream`
    - `BufferedReader / BufferedWriter`
- Both sets of APIs utilize `Exception`s

FileReader and FileWriter Example

Example 10-1: Copying a File

```
import java.io.*;
public class Ex10_1 {
  public static void main(String[] args) {
    int count = 0;
    try {
      FileReader in = new FileReader("filesource");
      FileWriter out = new FileWriter("filesink");
      int c;
      while ((c = in.read()) != -1){
        count++;
      }
      out.write(c);
      in.close();
      out.close();
    } catch (IOException e) {
      System.err.println(e.getMessage());
    }
    System.out.println("Copied " + count + " characters");
  }
}
```

StringReader and FileWriter Example

Example 10-2: Copying a string to a file

```
import java.io.*;
public class Ex10_2 {
  public static void main(String[] args) throws IOException {
    String s =
      "This is the string source to be used for input.";
    StringReader in = new StringReader(s);
    FileWriter out = new FileWriter("filesink");
    int c;
    int count = 0;
    while ((c = in.read()) != -1){
       count++;
     }
    out.write(c);
    in.close();
    out.close();
    System.out.println("Copied "+count+" bytes");
  }
}
```

Decorator Pattern

- Earlier we discussed adding functionality to an object through sub-classing
- The I/O package has many classes, creating subclasses for every permutation of functionality would be hard, time consuming, and not very robust
- So, I/O in Java utilizes an object oriented design pattern
  - Called the *Decorator Pattern*
  - *Decorator Pattern* adds functionality to an object by *wrapping* it instead of sub-classing it
  - Decorator objects can be wrapped by other decorator objects that can be wrapped by other decorator objects . . .
  - If you can find the functionality you need in a low-level API, wrap it with a decorator

Filter Streams and the Decorator Pattern

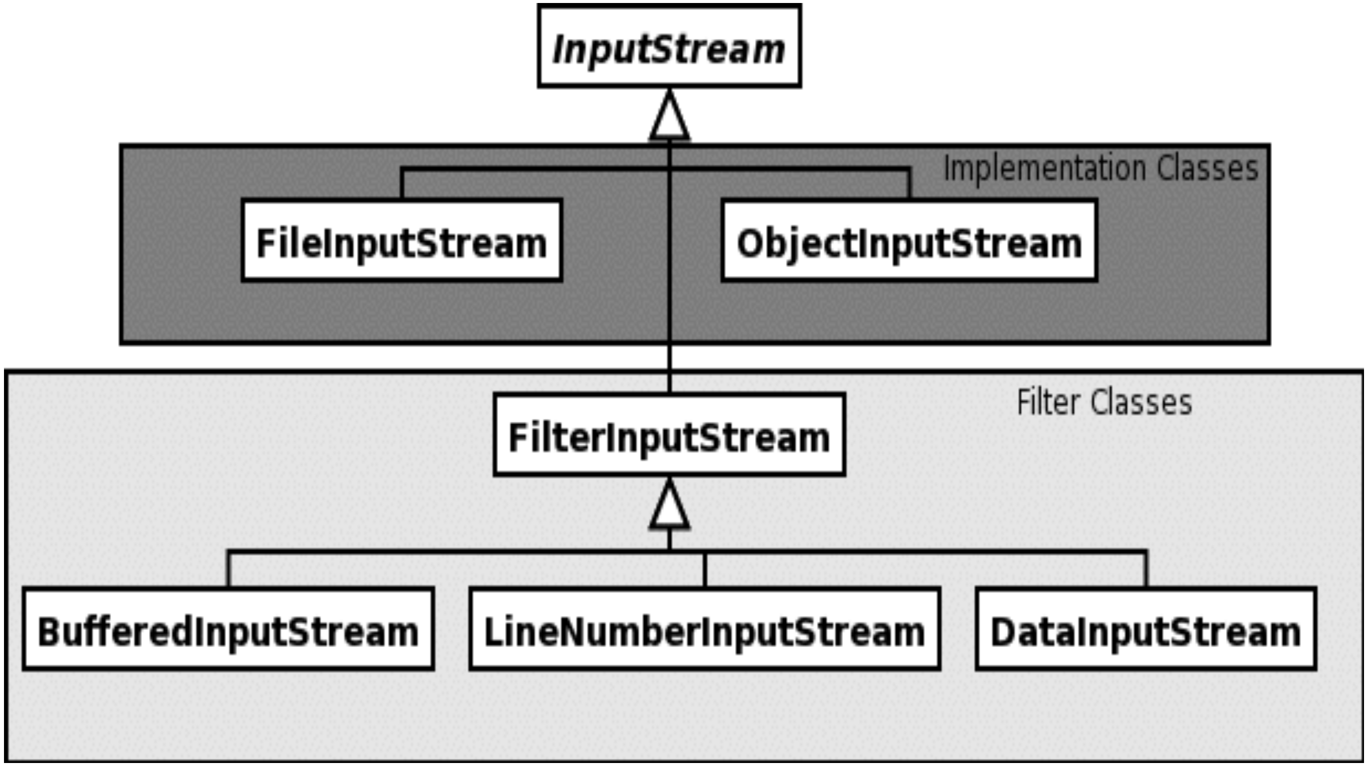
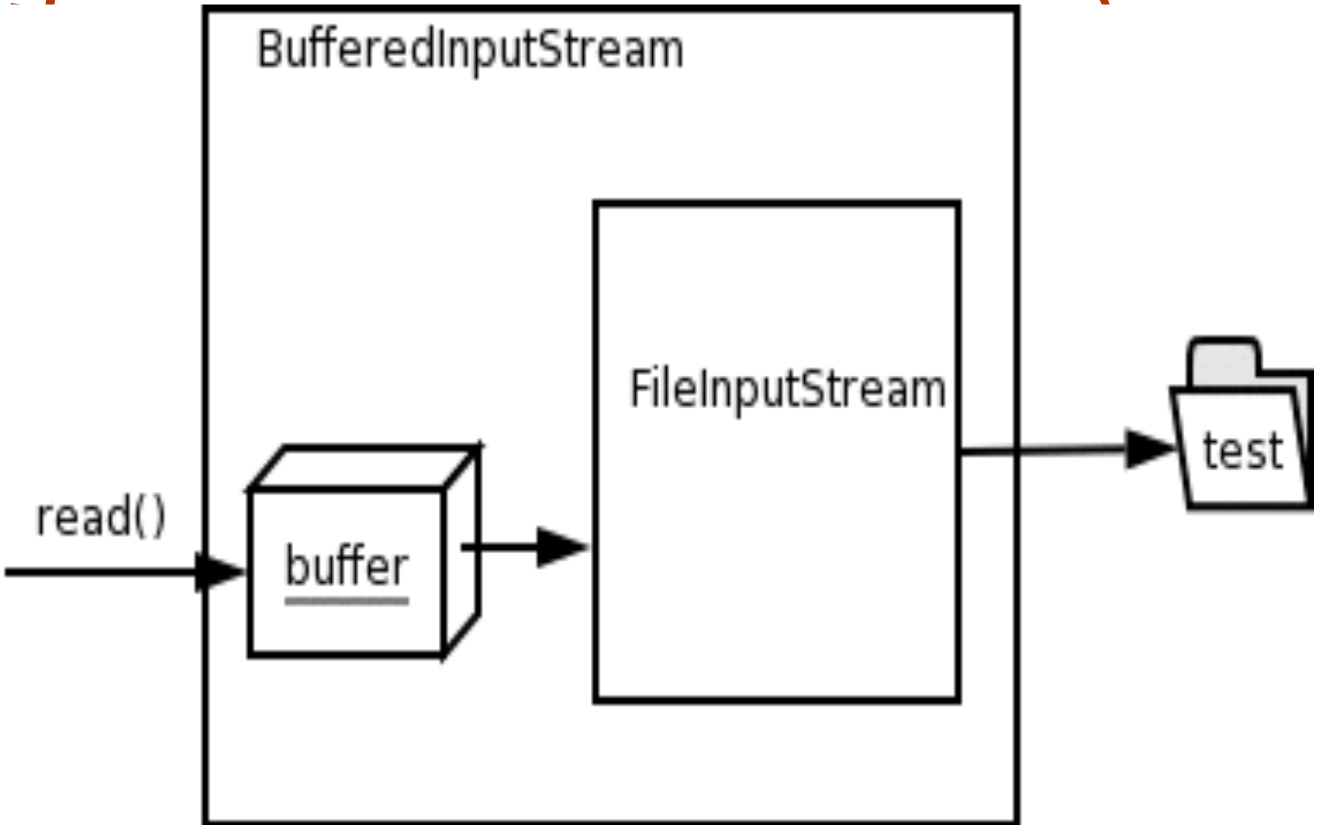


Fig 10-3: **Part of the stream hierarchy**

Using the Decorator Pattern

- Create a reference to the low-level stream
- Pass that reference to the decorator upon construction
- Call I/O operations on the decorator
  - Decorator initially performs operation
  - Decorator delegates operation to low-level stream
  - Low-level stream interactions are hidden from you and taken care of for you

Using the Decorator Pattern (cont.)



10-4: **A decorated FileInputStream**

Filter Class Example

Example 10-3. Using a BufferedReader

```
import java.io.*;
public class Ex10_3 {
  public static void main(String[] args) {
    try {
      BufferedReader input = new BufferedReader(
                                 new FileReader("TestInput.text"));
      String inputLine = new String();
      System.out.println("File output...");
      while((inputLine = input.readLine()) != null){
        System.out.println(inputLine);
      }
      input.close();
    } catch (IOException e) {
      System.err.println(e);
    }
  }
}
```

Converting Streams

- The two categories of streams are
  - Binary – used with binary data
  - Character – used with Unicode character data
- Classes in the binary category have a different inheritance tree than those of the character category
- Therefore you can't cast a binary stream into a character stream
- If you can't cast it, convert it!
- Two conversion utility classes
  - `InputStreamReader`
  - `OutputStreamWriter`

Stream Conversion Example

Example 10-4. byte stream to character stream

```
import java.io.*;
public class Ex10_4 {
  public static void main(String[] args) {
    // Create the Reader
    Reader r = new InputStreamReader(System.in);
    // Create the Buffered Reader
    BufferedReader input = new BufferedReader(r);
    try {
      while (true) {
        System.out.print("Enter a line ('end' terminates):");
        String s = input.readLine();
        if (s.equals("end"))
          break;
        System.out.println("You said -- " + s);
      }
      System.out.println("bye");
    } catch (Exception e) {
      System.err.println(e);
    }
  }
}
```

PrintWriter

- Writer is pretty low level
- FileWriter allows us to write to a file
- BufferedWriter makes FileWriter better
- But, what if we want to write out lines of text to a file in a single operation?
- Use PrintWriter!

Using a PrintWriter

Example 10-5. Using a PrintWriter

```
import java.io.*;
public class Ex10_5 {
   public static void main(String[] args) {
       // Create the Buffered Reader
      BufferedReader input = new BufferedReader(
          new InputStreamReader(System.in));
      try {
          // Create the writer (buffered!)
          PrintWriter pw = new PrintWriter(
                             new BufferedWriter(
                               new FileWriter("dialog.text")));
         pw.println("----------- Starting");
         int lineNum = 1;
. . .
```

Using a PrintWriter (cont.)

Example 10-5. Using a PrintWriter (continued)

```
            while (true) {
              System.out.print("Enter a line ('end' terminates):");
              String s = input.readLine();
              if (s.equals("end")) {
                break;
              }
              System.out.println("You said -- " + s);
              pw.print(lineNum++);
              pw.println(")  "+s);
            }
            System.out.println("bye");
            pw.println("----------- Done");
            pw.close();
        } catch (Exception e) {
          System.err.println(e);
        }
    }
}
```

Writing Java Data

The I/O API provides two sets of classes for reading and writing Java specific data

- `DataInputStream` / `DataOutputStream`
  - Used for reading / writing primitive data
  - Method for each primitive type
  - Preserves platform independence
  - Can be used to persist state of an object, but done very manually
- `ObjectInputStream` / `ObjectOutputStream`
  - Used for reading / writing objects
  - Utilizes *Object Serialization*
  - Easiest way to persist state of an object

DataStreams Example

Example 10-6. Using DataStreams

```
import java.io.*;
public class Ex10_6 {
  public static void main(String[] args) {
    try {
      DataOutputStream out = new DataOutputStream(
                               new BufferedOutputStream(
                                new FileOutputStream("Data.tmp")));
      out.writeDouble(839.829);
      System.out.println("Wrote " + 839.829);
      out.writeInt(-1872);
      System.out.println("Wrote " + (-1872));
      out.close();
 . . .
```

DataStreams Example (cont.)

Example 10-6. Using DataStreams (continued)

```
      DataInputStream in = new DataInputStream(
                                  new BufferedInputStream(
                                    new FileInputStream("Data.tmp")));
      double d = in.readDouble();
      System.out.println("Read " + d);
      int i = in.readInt();
      System.out.println("Read " + i);
      in.close();
    } catch (Exception e) {
      //bad practice.. But quick and dirty
      throw new RuntimeException(e);
    }
  }
}
```

Object Serialization

- Introduced as part of the Java Beans specification
- Complex mechanism to persist and restore the state of an object
  - Utilizes something referred to as object graphs
  - Objects within objects within objects are all stored
- Two types
  - Automatic
    - Follow some rules
    - `implement java.io.Serializable`
    - Persistence and restoration are done for you
  - Manual
    - Do most everything yourself
    - `implement java.io.Externalizable`
    - Complete control

Automatic Serialization Rules

- Class should be `public`
- Instance variables you don’t want saved should be marked `transient`
- Should have a `public` no-arg constructor
- Must implement `java.io.Serializable`

Object Serialization

Example 10-7. Using DataStreams

```
import java.io.*;
class DataObject implements Serializable {
     private int id;
     public DataObject(int n) {
        id = n;
     }
     public String toString() {
        return " DataObject " + id;
     }
  }
```

Object Serialization

Example 10-7. Using DataStreams, continued

```
public class Ex10_7 implements Serializable {
   private DataObject[] objects = { new DataObject(981),
                                    new DataObject(3),
                                    new DataObject(-98)
                                  };
   private String id = "Container";
   public String toString() {
     String s = "";
     for (int i = 0; i < 3; i++){
       s += objects[i];
      }
      return id + " " + s;
   }

   public static void main(String[] args) {
      Ex10_7 c = new Ex10_7();
```

. . .

Object Serialization

Example 10-7. Using DataStreams, continued

```
        try {
          System.out.println("Created object");
          System.out.println(c);
          ObjectOutputStream out = new ObjectOutputStream(
                                       new FileOutputStream("somewhere"));
          out.writeObject(c);
          out.close();
          c = null; // object is now out of scope.
          System.out.println("Written and destroyed.");
          ObjectInputStream in = new ObjectInputStream(
                                        new FileInputStream("somewhere"));
          Ex10_7 newc = (Ex10_7) in.readObject();
          in.close();
          System.out.println("Read.");
          System.out.println("Recovered  object");
          System.out.println(newc);
        } catch(Exception e) {
          System.err.println(e);
        }
    } //end main
} //end class
```

File Interactions

- Java provides classes to work with underlying file systems in platform independent manner
- The `File` class allows you to create an object representation for a file
- A `File` instance is not the file itself; but allows you to
  - Find out information about the underlying file
  - Delete, rename, move the underlying file
  - Check permissions
  - Etc.

The File Class

Example 10-8. Using `File`

```
import java.io.*;
public class Ex10_8 {
  public static void main(String[] args) {
     // get the current path
     File pwd = new File(".");
     String[] dirList = pwd.list();
     for(int i = 0; i < dirList.length; i++) {
       System.out.println(dirList[i]);
     }
  }
}
```

The File Class

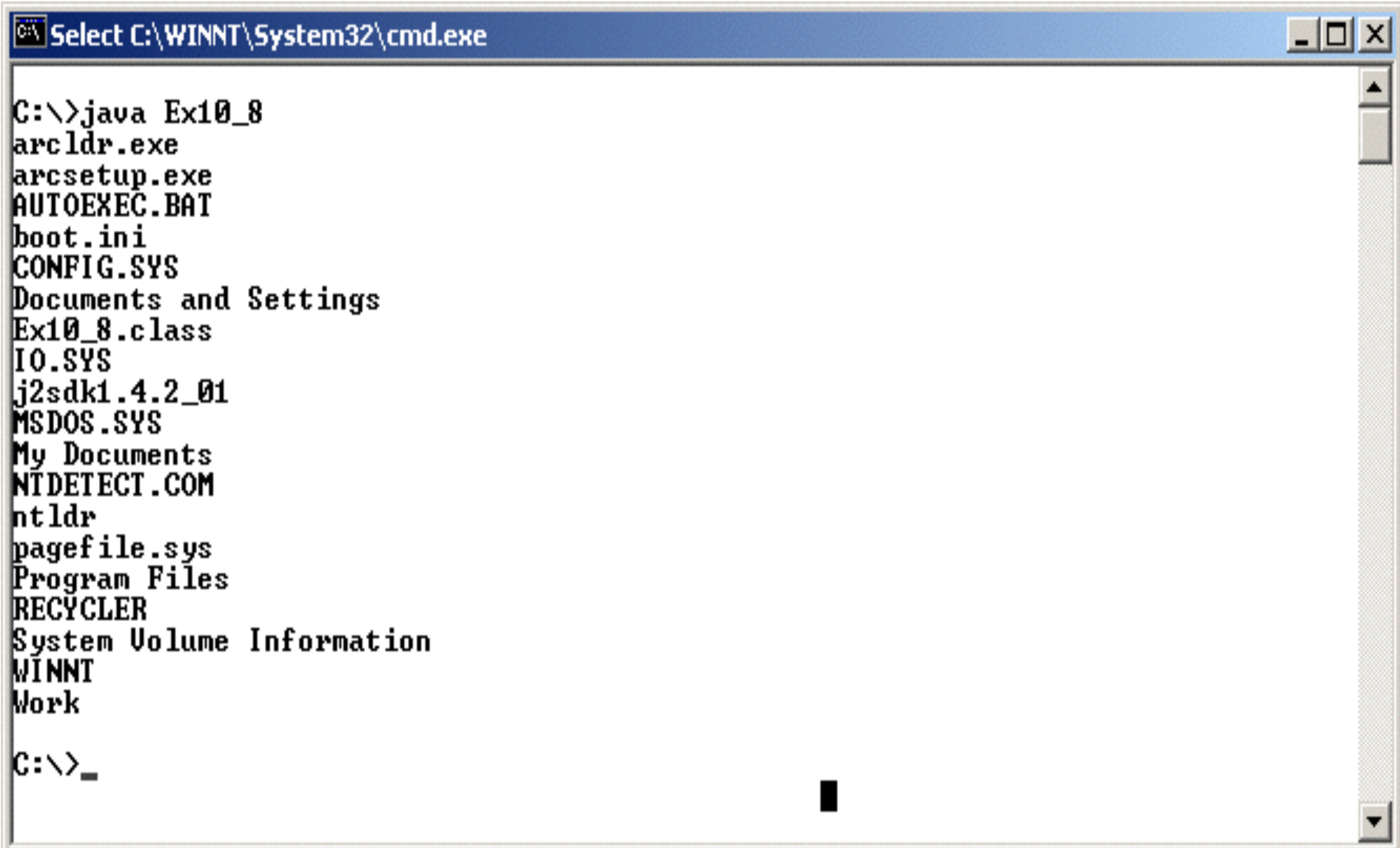


Fig 10-5: **Output from example 10-8**

Summary

We covered

- Describing the architecture of the `java.io` API
- Describing the streams model
- Using implementation streams
- Using filter streams
- Describing the difference between streams, readers and writers
- Using data streams and files
- Using buffered I/O and the `PrintWriter`
- Describing the `File` class

Collections
(Chpt. 11)

Objectives

At the end of this module you should be able to

- Describe the Collections Framework architecture
- Use an `Iterator`
- Use a `Set`
- Use a `List`
- Use a `Map`
- Use collection algorithms
- Use wrappers

Collections

- A collection is an object that plays the role of a container for other objects
- Arrays are considered collections
- Sometimes referred to as *data structures*
  - Linked Lists
  - Trees
  - Sets
  - Maps
  - Etc.

Collections Framework

- A set of APIs simplifying use of data structures
- Collections Framework API is composed of three main architectural components
  - **Interfaces**
    - Expose the functionality of collections to the programmer
    - Underlying container is manipulated through the interface independently of which data structure is being used in the implementation
    - Enables conversion between different implementations of data structures
  - **Implementations**
    - The data structures
    - Implement the functionality of the collection interfaces
  - **Algorithms and Wrappers**
    - Reusable functionality
    - Like sorting and searching

The Java Collections Framework Architecture

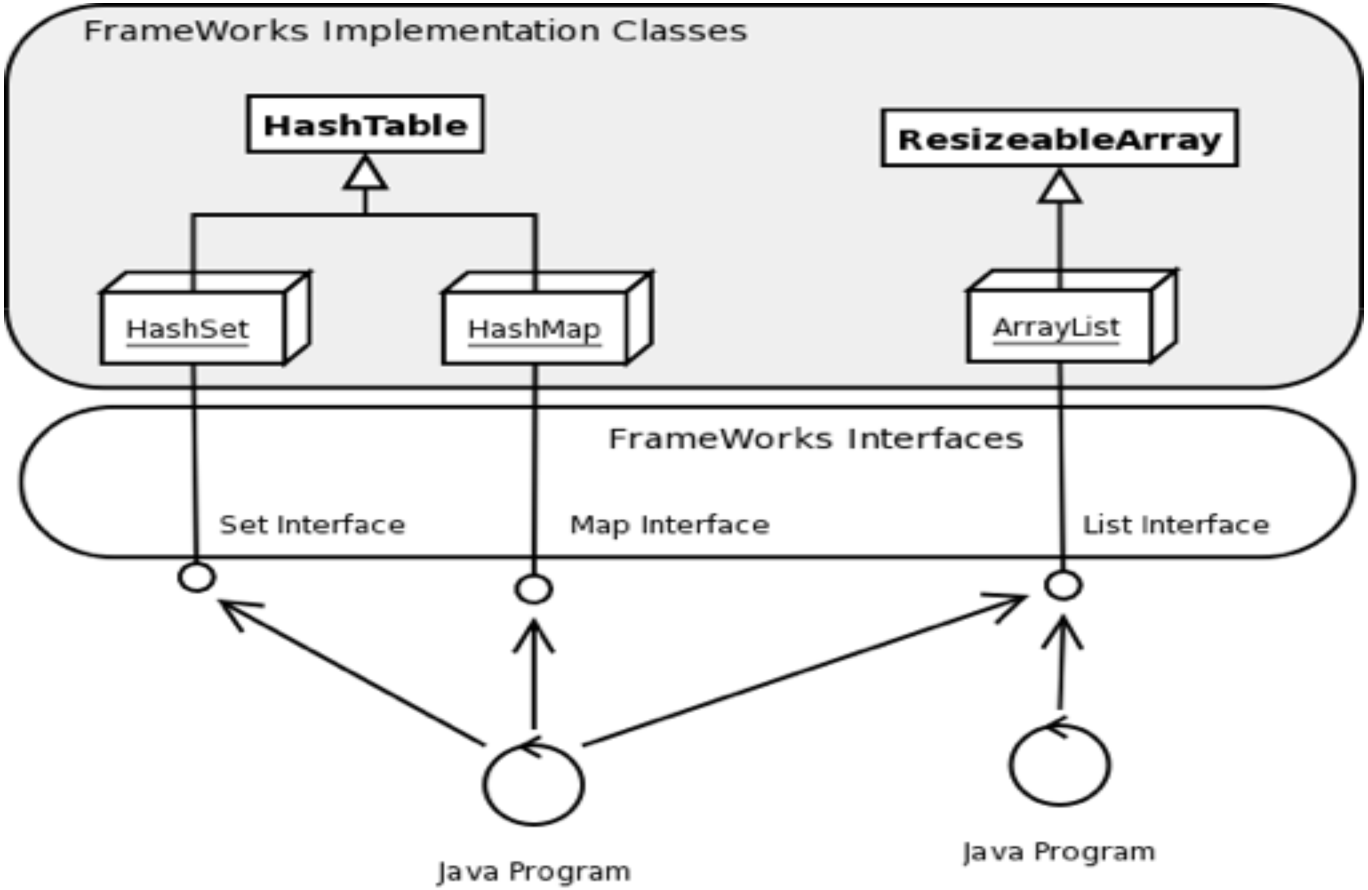


Fig. 11-1: Part of the Collections Framework architecture

Collection Types

Two main categories of collections

- `java.util.Collection`
  - Root interface in the *collection hierarchy*
  - May contain duplicates
  - May be ordered
  - Useful only through sub-interface implementations like
    - `ArrayList`
    - `HashSet`
- `java.util.Map`
  - An object that maps keys to values
  - Cannot contain duplicate keys
  - Each key can map to at most one value
  - Useful only through sub-interface implementations
    - `TreeMap`
    - `HashMap`

The Collections Interfaces

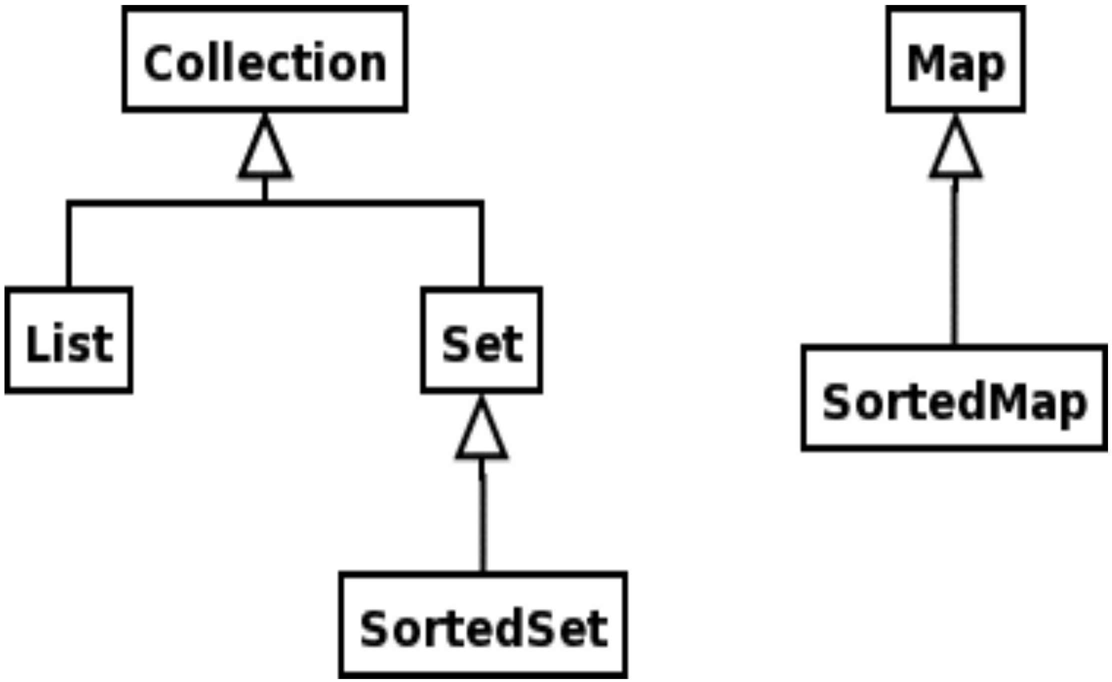


Fig. 11-2: Frameworks Interface Hierarchy

Implementation Classes

- The implementation classes are derived from four basic data structures
  - hash table
  - resizable array
  - balanced tree
  - linked list
- We can’t cover data structures . . . So let’s just look at the implementation options

Implementation Class Chart

Interfaces	Container Types			
	Hash Table	Resizable Array	Balanced Tree	Linked List
Set	HashSet		TreeSet	
List		ArrayList		LinkedList
Map	HashMap		TreeMap	

Fig. 11-3: Frameworks implementation classes

Collection Interface API

```
public interface Collection {
   // Basic Operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(Object element);     // Optional
    boolean remove(Object element);  // Optional
    Iterator iterator();

   // Bulk Operations
    boolean containsAll(Collection c);
    boolean addAll(Collection c);              // Optional
    boolean removeAll(Collection c);           // Optional
    boolean retainAll(Collection c);           // Optional
    void clear();                              // Optional

   // Array Operations
    Object[] toArray();
    Object[] toArray(Object a[]);
}
```

Using the Collections Framework

Basic steps for using collections framework

1. Select the interface appropriate for the application
2. Select the desired data structure implementation
3. Instantiate the implementation
4. Manipulate the data structure using the interface

Creating, Filling & Printing Collections Example

Example 11-1: Creating, filling and printing collections

```
import java.util.*;
// This is a utility class that provides a method for
//  filling a collection -- any collection because it only uses
//  the methods in the collection interface. This shows the
//  use of the Collections type as a general type for passing
// as an argument.
class Fill {
  static Collection init(Collection c, int slots) {
    for (int i = 0; i < slots; i++) {
      c.add("Test Value " + i);
    }
    return c;
  }
}
```

Creating, Filling and Printing Collections (cont.)

Example 11-1: Creating, filling and printing collections (continued)

```
public class Ex11_1 {
  public static void main(String[] args) {
    Collection arrayList = new ArrayList();
    Collection hashSet = new HashSet();
    Collection treeSet = new TreeSet();
    Collection linkList = new LinkedList();
    arrayList = Fill.init(arrayList,5);
    hashSet = Fill.init(hashSet,5);
    treeSet = Fill.init(treeSet,5);
    linkList = Fill.init(linkList,5);
    System.out.println("ArrayList");
    System.out.println(arrayList);
    System.out.println("HashSet");
    System.out.println(hashSet);
    System.out.println("TreeSet");
    System.out.println(treeSet);
    System.out.println("LinkedList");
    System.out.println(linkList);
  }
}
```

Creating, Filling and Printing Collections Output

Example 11-1: Creating, filling and printing collections (continued)

```
// Output is
ArrayList
[Test Value 0, Test Value 1, Test Value 2, Test Value 3, Test Value 4]
HashSet
[Test Value 2, Test Value 3, Test Value 1, Test Value 0, Test Value 4]
TreeSet
[Test Value 0, Test Value 1, Test Value 2, Test Value 3, Test Value 4]
LinkedList
[Test Value 0, Test Value 1, Test Value 2, Test Value 3, Test Value 4]
```

Iterator Interface API

- Both `java.util.Collection` and `java.util.Map` provide a mechanism to iterate over the contained values
- `Iterator` is an interface describing how to `Iterator` over the collection
- Each implementation class will provide its own `Iterator` implementation

```
public interface Iterator {
    boolean hasNext();
    Object next();
    void remove();    // Optional
}
```

Iteration Example

Example 11-2: Iterating over a collection, removing even elements

```
import java.util.*;
// Now we have added a generic Iterator method
class Fill {
   static Collection init(Collection c, int slots) {
     for (int i = 0; i < slots; i++) {
       c.add("Test Value " + i);
     }
     return c;
   }
   static void deleteSecond(Collection c) {
     Iterator itr = c.iterator();
     boolean even = false;
     while (itr.hasNext()) {
       itr.next();
       if (even) {
        itr.remove();
       }
       even = !even;
     }
   }
}
```

Iteration Example (cont.)

Example 11-2: Iterating over a collection, removing even elements (continued)

```
public class Ex11_2 {
   public static void main(String[] args) {
     Collection arrayList = new ArrayList();
     Collection hashSet = new HashSet();
     Collection treeSet = new TreeSet();
     Collection linkList = new LinkedList();
     arrayList = Fill.init(arrayList, 5);
     hashSet = Fill.init(hashSet, 5);
     treeSet = Fill.init(treeSet, 5);
     linkList = Fill.init(linkList, 5);
     System.out.println("ArrayList");
     Fill.deleteSecond(arrayList);
     System.out.println(arrayList);
     System.out.println("HashSet");
     Fill.deleteSecond(hashSet);
     System.out.println(hashSet);
     System.out.println("TreeSet");
     Fill.deleteSecond(treeSet);
     System.out.println(treeSet);
     System.out.println("LinkedList");
     Fill.deleteSecond(linkList);
     System.out.println(linkList);
   }
 }
```

Iteration Example Output

Example 11-2: Iterating over a collection, removing even elements (continued)

```
// Output is
ArrayList
[Test Value 0, Test Value 2, Test Value 4]
HashSet
[Test Value 2, Test Value 1, Test Value 4]
TreeSet
[Test Value 0, Test Value 2, Test Value 4]
LinkedList
[Test Value 0, Test Value 2, Test Value 4]
```

Set Interface

- Interface models the mathematical *set* abstraction
- A sub-interface of `java.util.Collection`
- A collection that contains no duplicate elements

Set Interface Example

Example 11-4: Set Interface

```
import java.util.*;
class Test{}  // something to put in the Set
public class Ex11_4 {
  public static void main(String [] args) {
    Set s = new HashSet();  // create the set
    Test t = new Test();
    s.add(t);
    s.add(t);     // duplicate entry
    s.add("One");
    s.add("Two");
    s.add("One");
    s.add("One");
    s.add("Three");
    s.add("Four");
    s.add("Four");
    s.add("Four");
    s.add(new Test());  /// not a duplicate
    System.out.println(s);
  }
}
// Output is:
[Test@107077e, Test@11a698a, Four, Three, Two, One]
```

List Interface

- Interface models an ordered collection, or *sequence*
- A sub-interface of `java.util.Collection`
- A collection contain duplicate elements
- Implementations typically allow `null`
- Supports positional access for insertion and retrieval (based on index)
- Has a special type of `Iterator`, `ListIterator`
  - Allows insertion and replacement while iterating over the collection
  - Supports `Iterator` interface operations

List Interface API

```
public interface List extends Collection {
    // Positional Access
    Object get(int index);
    Object set(int index, Object element);            // Optional
    void add(int index, Object element);              // Optional
    Object remove(int index);                         // Optional
    abstract boolean addAll(int index, Collection c); // Optional

    // Search
    int indexOf(Object o);
    int lastIndexOf(Object o);

    // Iteration
    ListIterator listIterator();
    ListIterator listIterator(int index);

    // Range-view
    List subList(int from, int to);
}
```

List Iterators API

```
public interface ListIterator extends Iterator {
    boolean hasNext();
    Object next();
    boolean hasPrevious();
    Object previous();
    int nextIndex();
    int previousIndex();

    void remove();           // Optional
    void set(Object o);      // Optional
    void add(Object o);      // Optional
}
```

List Example

Example 11-5: Working with a List Iterator

```
import java.util.*;
public class Ex11_5 {
  public static void main(String[] args) {
    List L = new LinkedList();
    for (int i = 0; i < 10; i++) {
      L.add("" + i);
    }
    System.out.println("List created");
    System.out.println(L);
    L.add(4, "10");
    System.out.println(L);
    L.set(5, "11");
    System.out.println(L);
    ListIterator itl = L.listIterator(4);
    System.out.println("L[4]=" + L.get(4));
    itl.previous();
    itl.remove();
    System.out.println(L);}
  }
}
```

```
// output
List created
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 10, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 10, 11, 5, 6, 7, 8, 9]
L[4]=10
[0, 1, 2, 10, 11, 5, 6, 7, 8, 9]
```

Map Interface

- Interface models the mapping of keys to values
- A collection that contains no duplicate elements
- No `Iterator` functionality
- Provides three *views* of data that allow us to obtain `Iterator`s
  - Keys
  - Values
  - Entry set (key-value mappings)

Map Interface API

```
public interface Map {
    // Basic Operations
    Object put(Object key, Object value);
    Object get(Object key);
    Object remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    // Bulk Operations
    void putAll(Map t);
    void clear();

    // Collection Views
    public Set keySet();
    public Collection values();
    public Set entrySet();

    // Interface for entrySet elements
    public interface Entry {
        Object getKey();
        Object getValue();
        Object setValue(Object value);
    }
}
```

Map Example

Example 11-6: Map Interface

```
public class Ex11_6 {
  public static void main(String[] args) {
    Map custs = new HashMap();
    custs.put("982098", new Customer("Bill White"));
    custs.put("116201", new Customer("Bob Green"));
    custs.put("983611", new Customer("Saj Black"));
    custs.put("661109", new Customer("Sharon Brown"));
    System.out.println(custs);
    // Some typical Map operations
    custs.remove("116201");
    custs.put("761102", new Customer("Simone Blanc"));
    System.out.println(custs.get("661109"));
    System.out.println(custs);
  . . .
```

Map Example (cont.)

Example 11-6: Map Interface (continued)

```
    // Now we walk through the entries
    Set entries = custs.entrySet();
    Iterator iter = entries.iterator();
    while (iter.hasNext()) {
      Map.Entry entry = (Map.Entry) iter.next();
      Object key = entry.getKey();
      Object value = entry.getValue();
      System.out.println("key=" + key + ", value=" + value);
    }
  } //end main
} //end class
```

java.util.Collections

- A utility class that provides
  - Algorithms
  - Wrappers
- Are static methods contained in the
- Common algorithms for things like
  - Binary search
  - Reversing
  - Shuffling
  - Sorting
- Wrappers for creating
  - Singletons
  - Synchronized collections
  - Unmodifiable collections

Algorithm Example

Example 11-7: Using Algorithms

```
public class Ex11_7 {
  public static void main(String[] args) {
     // create a list
     List numbers = new ArrayList(20);
     for (int i = 1; i <= 20; i++) {
       numbers.add(new Integer(i));
     }
     System.out.println("Starting List");
     System.out.println(numbers);
       // Now randomize
     Collections.shuffle(numbers);
     System.out.println("Shuffled List");
     System.out.println(numbers);
      // Now sort
     Collections.sort(numbers);
     System.out.println("Sorted List");
     System.out.println(numbers);
   }
}
```

Algorithm Example (cont.)

Example 11-7: Using Algorithms (continued)

```
// output
Starting List
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
Immutable List
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
Exception in thread "main" java.lang.UnsupportedOperationException
    at java.util.Collections$UnmodifiableCollection.add(Collections.java:1004)
    at Ex11_8.main(Ex11_8.java:17)
```

Wrapper Example

Example 11-8: Using a Wrapper

```
import java.util.*;
public class Ex11_8 {
  public static void main(String[] args) {
     // create a list
    Collection numbers = new ArrayList(20);
    for (int i = 1; i <= 20; i++){
      numbers.add(new Integer(i));
    }
    System.out.println("Starting List");
    System.out.println(numbers);

     // Make the list immutable using the static method
    numbers = Collections.unmodifiableCollection(numbers);
    System.out.println("Immutable List");
    System.out.println(numbers);

    // Try to add a new number
    numbers.add(new Integer(100));}
  }
}
```

Summary

We covered

- Describing the Collections Framework architecture
- Using an `Iterator`
- Using a `Set`
- Using a `List`
- Using a `Map`
- Using an algorithm
- Using wrappers

Multi-Threading
(Chpt. 12 - Optional)

Objectives

At the end of this module you should be able to

- Understand concurrency and threads, and why they are used
- Create and run a thread using the `Thread` class
- Understand thread priorities and scheduling
- Describe what a daemon thread is
- Use the `Runnable` interface
- Use thread synchronization
- Understand how threads are coordinated

Threads

- **Process:** A flow of control running with its own address space, stack, etc.
- **Thread:** A flow of control sharing an address space with another thread, but with its own stack, registers, etc.
- **Concurrency:** Processes that are running at the same time

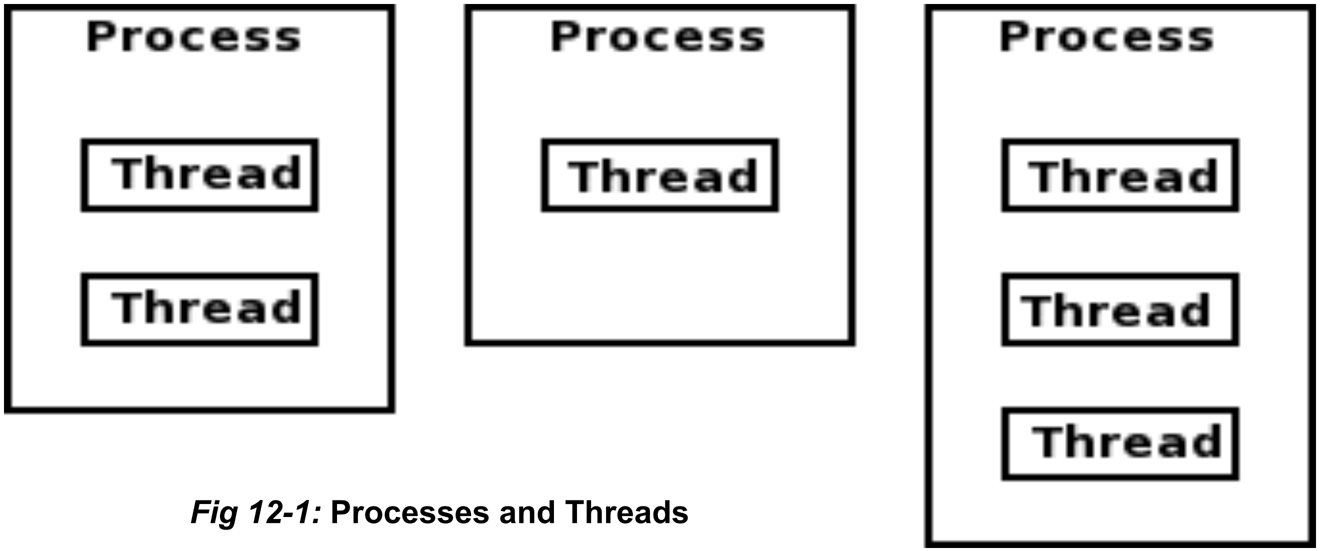


Fig 12-1: **Processes and Threads**

Threads in the Language

- Java as a language has built in support for multi-threaded applications
- There two primary classes used in threaded programming
  - `java.lang.Thread`
    - Class that represents a Thread
    - Two main lifecycle methods
      - `start` – prepares the thread for execution
      - `run` – main body of execution
    - Basically an “empty” `Thread`; `run` has no useful implementation
  - `java.lang.Runnable`
    - Interface for creating a body of execution for a `Thread`
    - Has one method – `run`
    - Threads delegate execution to `Runnable`’s `run`

Threads in the Language (cont.)

There are two ways to utilizes Threads

- `extend java.lang.Thread`
  - Over-ride run method to do something useful
  - Consider supplying application specific constructors
  - Consider building `Thread` management mechansim
  - Once a `Thread` instance finishes, it can not be reused
- `implement java.lang.Runnable`
  - Any object can become the body of execution for a `Thread`
  - More flexible
  - A `Runnable` instance can be reused

Threads in the Platform

- Java as a platform has built in support for multi-threaded applications
- The virtual machine has its own *thread scheduler*
  - Provides platform independent thread scheduling
  - Usually maps down to or interacts with underlying OS scheduler
- The JVM uses pre-emptive scheduling
  - Highest-priority thread is always running
  - Pre-emption may be effected by underlying OS scheduler

Extending Thread Example

Example 12-1: Spawning Threads

```
public class Ex12_1 extends Thread {
  private int iterations = 0;  // loop counter
  private int id;              // number of thread
  private static int threadNumber = 1;
  public Ex12_1 () {
    id = threadNumber++;
    start();
  }
  public void run() {
    while (iterations++ <3) {
      try {
        sleep(5);
    } catch (InterruptedException e) {}
    System.out.println("Thread "+id+": iteration "+iterations );}
  }
  public static void main(String[] args) {
    for(int i = 0; i < 4; i++){
      new Ex12_1();
    }
  }
}
```

Extending Thread Example Output

C:\WINNT\System32\cmd.exe

```
C:\Work>java Ex12_1
Thread 1: iteration 1
Thread 2: iteration 1
Thread 3: iteration 1
Thread 1: iteration 2
Thread 2: iteration 2
Thread 3: iteration 2
Thread 4: iteration 1
Thread 1: iteration 3
Thread 2: iteration 3
Thread 3: iteration 3
Thread 4: iteration 2
Thread 4: iteration 3

C:\Work>_
```

Fig 12-2: **Output from example 12-1**

Thread Control

- Since Java is pre-emptive, you may need to do some work to avoid thread starvation
- `java.lang.Thread` has methods that can help
  - `sleep`
  - `Yield`
- There are no methods to `stop`, `suspend`, or `resume` a `Thread`’s execution; they have all been *deprecated* (eventually will be removed)

Thread Control (cont.)

- Adjusting a `Thread`'s priority can also be useful to prevent starvation
  - Integral values that range between the values provided in the `Thread` class
  - Range represented as constants
    - `MIN_PRIORITY`
    - `MAX_PRIORITY`
  - Range usually falls between 1 and 10 with default value of 5
  - Thread scheduler in JVM is responsible for keeping priority settings platform independent
- Consider creating daemon threads
  - Like daemon processes in an operating system
  - Run continuously in the background
  - Method `setDaemon()` is called before the `start()`
  - Cannot start any non-daemon threads

Thread Yielding Example

Example 12-2: Yielding Threads

```
public class Ex12_2 extends Thread {
  private int iterations = 0;   // loop counter
  private int id;               // number of thread
  private static int threadNumber = 1;   // Next available thread number
  public Ex12_2 () {
     id = threadNumber++;
  }
  public void run() {
    while (iterations++ <3) {
      System.out.println("Thread "+id+": iteration "+iterations );
      yield();
    }
  }
  public static void main(String[] args) {
    for(int i = 0; i < 4; i++){
      Ex12_2 t =new Ex12_2();
      t.start();
    }
  }
}
```

Thread Yielding

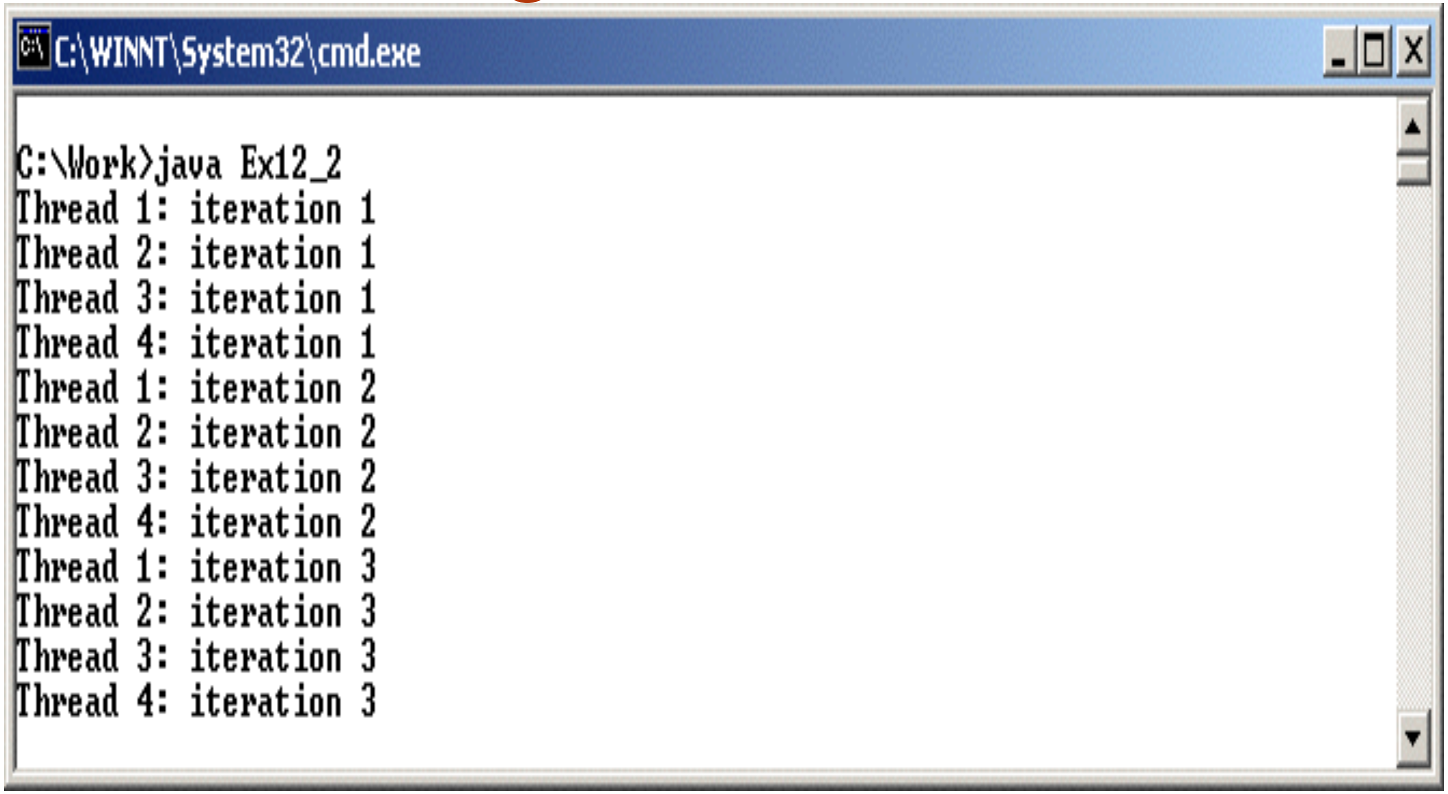


Fig 12-3: **Output from example 12-2**

Thread State Transitions

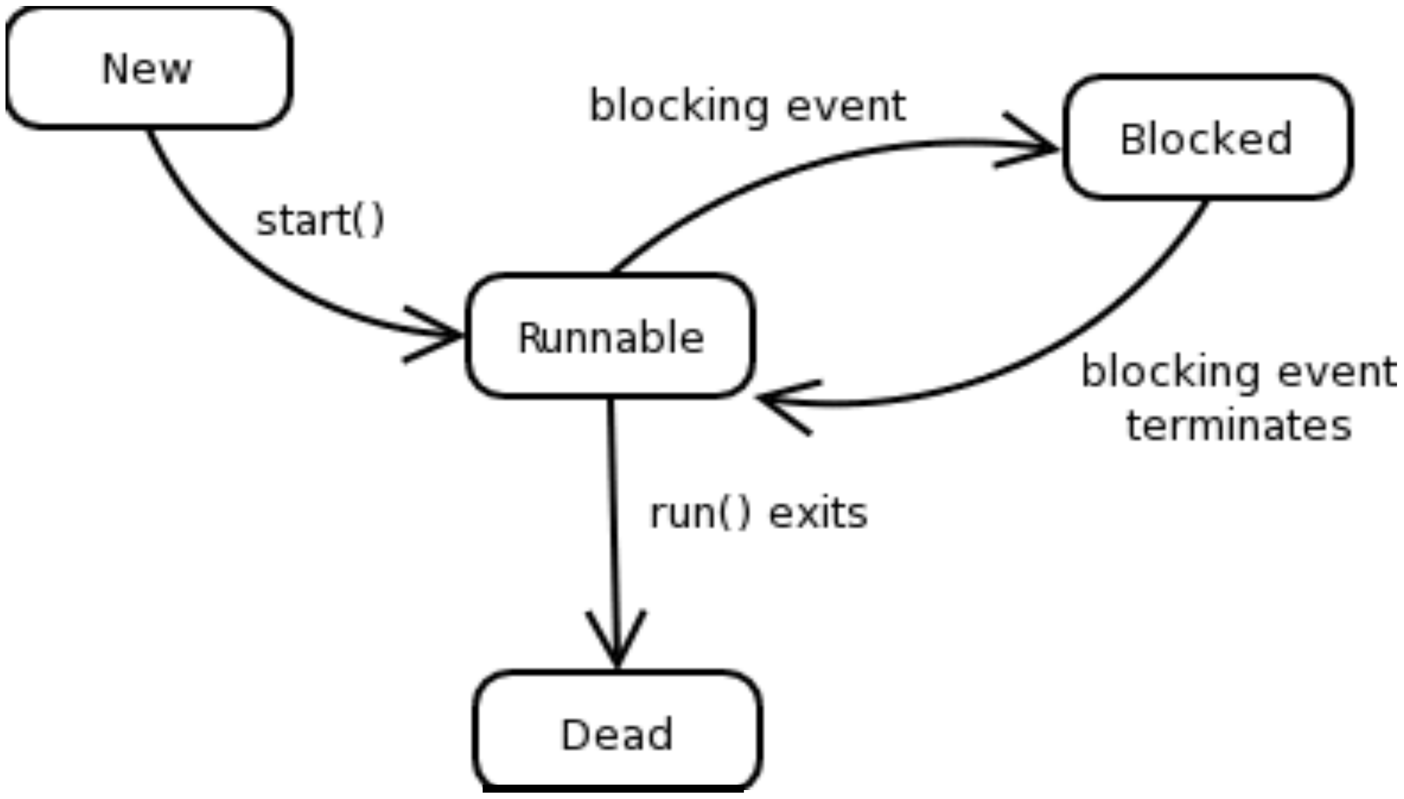


Fig 12-4: **Thread state transitions**

Interrupting Threads Example

Example 12-3: Interrupting Threads

```
// This is the thread that blocks
class BlockedThread extends Thread {
  public BlockedThread() {
    start();
  }
  public void run() {
    try {
      System.out.println("BlockedTread running...");
      synchronized(this) {
        System.out.println("BlockedTread blocking...");
        wait();
      }
    } catch(InterruptedException e) {
      System.out.println("BlockedTread Interrupted");
    }
  }
}
```

Interrupting Threads Example (cont.)

Example 12-3: Interrupting Threads (continued)

```
// This is the thread that interrupts
public class RudeThread extends Thread{
  // Get a Blocked thread to work with..
  static BlockedThread blocked = new BlockedThread();
  // Run the code.
  public static void main(String[] args) {
    RudeThread c = new RudeThread();
    c.start();
  }
  public void run() {
    try {
      sleep(5000);
    } catch (InterruptedException e) {}
    System.out.println("Preparing to interrupt ");
    blocked.interrupt();
    blocked = null;
  }
}
```

Runnable Interface Example

Example 12-4: The Runnable Interface

```
public class Ex12_4 implements Runnable {
  public void run() {
    System.out.println(" Main Method in Runnable Object " +
                                this);
  }
  public static void main(String[] args) {
    Thread t1 = new Thread(new Ex12_4());
    t1.start();
    System.out.println(" Thread Object " + t1);
   }
 }
```

Thread Access Control

- Deals with *synchronization* of Threads more than avoiding starvation
- There are two ways to create Thread synchronization
  - `synchronized` methods
  - `synchronized` blocks
- Synchronization relies on obtaining and freeing object locks
- Object locks are obtained when a thread executes `synchronized` code
- `java.lang.Object` has built-in mechanisms for notifying other `Thread` about lock status
  - `wait`
  - `notify`
  - `notifyAll`

Synchronized Method Example

Example 12-6: Synchronizing the deposit method

```
class Account {
  PrintWriter out;
  Account(PrintWriter p) {
    out = p;
  }
  synchronized void deposit(int amount, String name) {
    int balance; // local copy of balance
    out.println(name + " trying to deposit " + amount);
     // update local copy
    balance = getBalance();
    balance += amount;
    setBalance(balance);
  }

// Output of this code is
#1 trying to deposit 1000
#2 trying to deposit 1000
*** Final balance is 2000
```

Synchronized Block Example

Example 12-7: Synchronizing a block of code

```
class Account {
  PrintWriter out;
  Account(PrintWriter p) {
    out = p;
  }
  void deposit(int amount, String name) {
    int balance; // local copy of balance
    out.println(name + " trying to deposit " + amount);
     // update local copy in synch block
    synchronized(this) {
       balance = getBalance();
       balance += amount;
       setBalance(balance);
    }
   }

// Output of this code is
#1 trying to deposit 1000
#2 trying to deposit 1000
*** Final balance is 2000
```

Thread Cooperation

Taken From Example 12-5

```
    // Start the threads
    first.start();
    second.start();
    // wait here until both threads complete
    try {
      first.join();
      second.join();
    } catch (InterruptedException e) {
  }

  // Print the final result
  out.println("*** Final balance is " + remoteBalance);
```

Summary

We covered

- Concurrency and threads, and why they are used
- Creating and running a thread using the `Thread` class
- Thread priorities and scheduling
- What a daemon thread is
- Using the `Runnable` interface
- Using thread synchronization
- How threads are coordinated

Java Networking
(Chpt. 13 – Optional)

Objectives

At the end of this module you should be able to

- Describe ports and sockets
- Describe clients and servers
- Write a `ServerSocket` class
- Write a client `Socket` class
- Write a multi-threaded `ServerSocket`
- Read from a URL

Identifying Hosts

Example 13-1: Looking up an IP address

```
import java.net.*;
public class Ex13_1 {
  public static void main(String[] args) {
    if(args.length != 1) {
       System.err.println("Usage: EX13_1 MachineName");
       System.exit(1);
     }
     try {
        InetAddress IPAddress = InetAddress.getByName(args[0]);
        System.out.println(IPAddress);
      } catch (UnknownHostException e) {
        System.out.println("No IP address found for " + args[0]);
      }
    }
 }
```

Clients & Server

- **Port**
  - Each server is assigned a unique port number to use where it listens for connection requests
  - By convention, the use of ports 1 through 1024 is restricted to the operating system and standardized services
- **Sockets**
  - A software object that is created to represent a connection between two machines
  - Anytime a client and server connect a socket object is created on each machine

Connecting to a Time Service

Example 13-2: Connecting to the time service

```
import java.io.*;
import java.net.*;
public class Ex13_2 {
public static void main(String[] args) {
  try {
    Socket s = new Socket("time-A.timefreq.bldrdoc.gov", 13);
    InputStream istrm = s.getInputStream();
    BufferedReader input = new BufferedReader(
                                new InputStreamReader(istrm));
    String line = null;
    do {
      line = input.readLine();
      if (line == null) {
        break;
      }
      System.out.println(line);
    } while (line != null);
    s.close();
  } catch (Exception e) {
    System.err.println(e.getMessage());}
  }
}
```

Writing a Server

1. A `ServerSocket` object is instantiated and listens at a specific port
2. A client program (it might not be a Java program) requests a connection
3. If the `ServerSocket` accepts the connection, then its `accept()` returns a `Socket` that will be the server end of the connection
4. The connection is established with a `Socket` object at each end
5. InputStream and `OutputStream` objects are acquired from both `Socket` objects over which the network data transfers will take place

Server Example

Example 13-3: Client-server using sockets – Server side

```
import java.net.*;
import java.io.*;
public class Ex13_3 {
public static void main(String[] args) throws IOException {
  ServerSocket server = new ServerSocket(8099);
  System.out.println("Server started: " + server);
  try {
    // accept() tells the server to listen.
    // Program blocks until a client asks for a connection
    Socket connection = server.accept();
    //  Now we have a connection and we can continue.
    try {
      System.out.println( "Connection established: "+ connection);
      //  Create the input and output streams
      BufferedReader input = new BufferedReader(
                              new InputStreamReader(
                                connection.getInputStream()));
      PrintWriter output = new PrintWriter(
                            new BufferedWriter(
                             new OutputStreamWriter(
                               connection.getOutputStream())),true);
      // We now loop until the client quits the connection
      . . .
```

Server Example (cont.)

Example 13-3: Client-server using sockets – Sever side (continued)

```
        while(true) {
          String s = input.readLine();
          if (s.equals("quit")) {
            break;
          }
          System.out.println("Client said: " + s);
          output.println("You said "+ s);
        }
      //  Make sure the system resources are released
      } finally {
        System.out.println("Closing connection...");
        connection.close();
      }
    } finally {
      System.out.println("Server shutdown...");
      server.close();
    }
  } //end main
} //end class
```

Client Example

Example 13-4: Client-server using sockets – Client side

```
import java.net.*;
import java.io.*;
public class Ex13_4 {
  public static void main(String[] args) throws IOException {
    InetAddress addr = InetAddress.getByName(null);
    Socket connection = new Socket(addr, 8099);
    // Make sure we clean up the sockets now that we have a
    // socket connection
    try {
      System.out.println("connection socket = " + connection);
      BufferedReader input =new BufferedReader(
      new InputStreamReader(connection.getInputStream()));
      PrintWriter output = new PrintWriter(
                            new BufferedWriter(
                             new OutputStreamWriter(
                              connection.getOutputStream())),true);
```

Client Example (cont.)

Example 13-4: Client-server using sockets – Client side (continued)

```
        for (int i = 0; i < 10; i++) {
          output.println("Client generated line " + i);
          String s = input.readLine();
          System.out.println(s);
        }
        // Now we quit the connection
       output.println("quit");
      } finally {
        //  return system resources
        System.out.println("Closing connection...");
        connection.close();
      }
    } //end main
  } //end class
```

Server Started Output

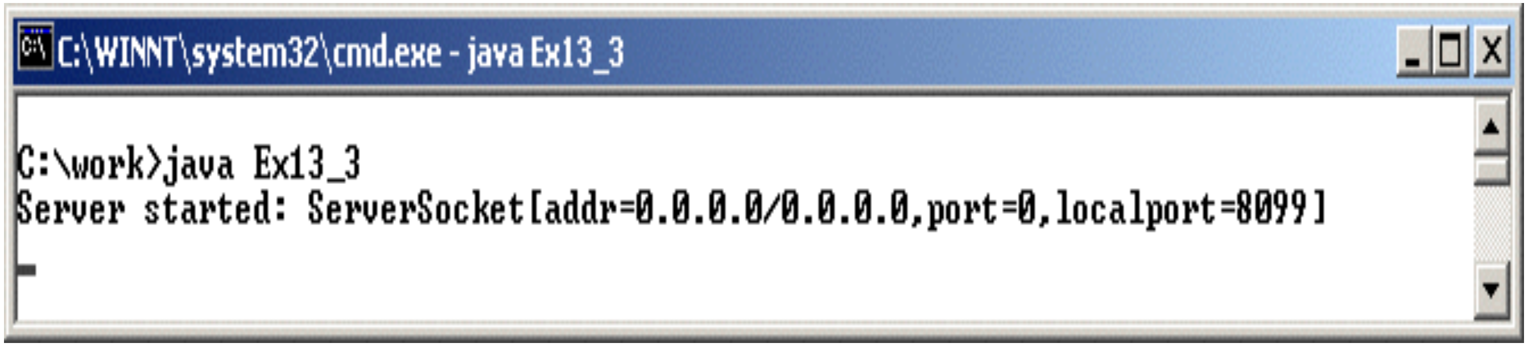


Fig 13-1: **Output from examples.**

Client Started Output

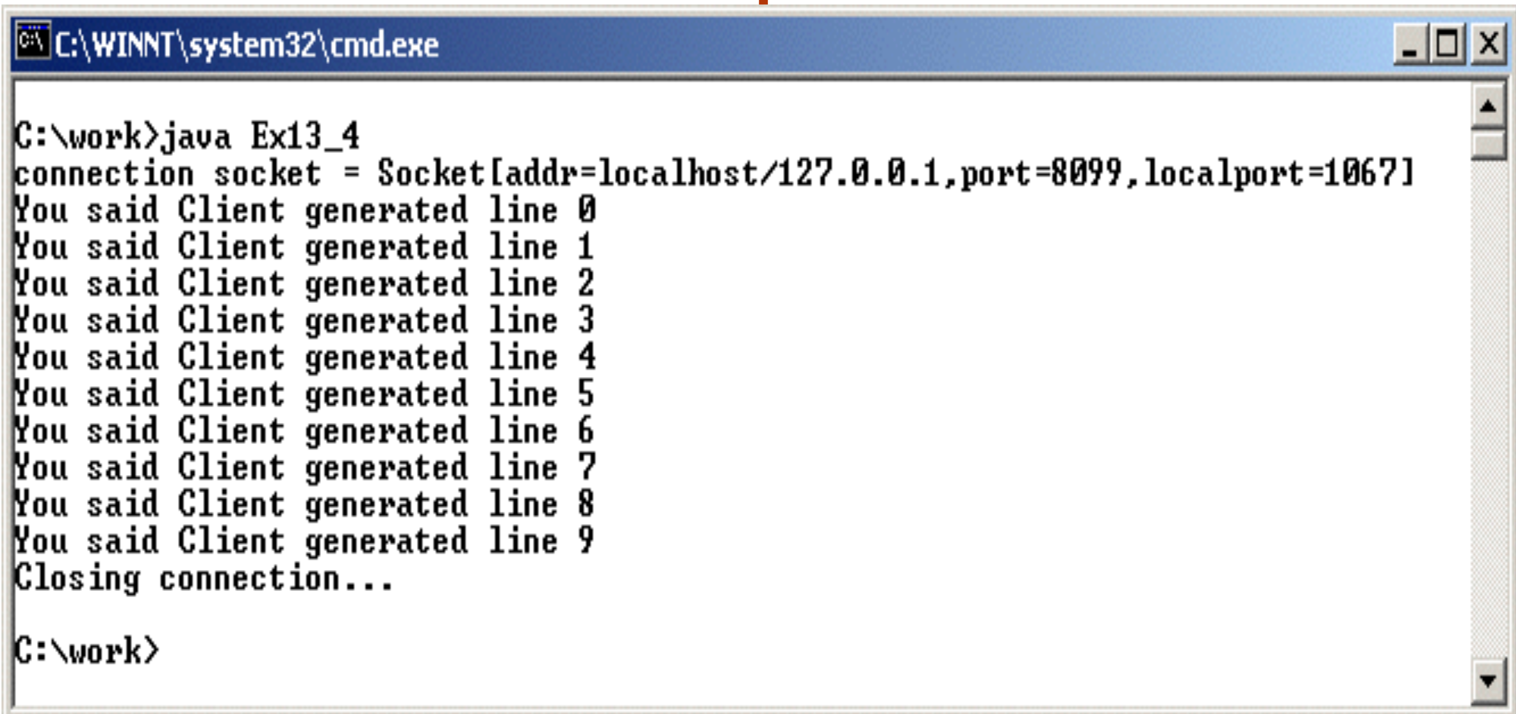


Fig 13-1: **Output from examples.**

Server Started & Connection Established

C:\WINNT\system32\cmd.exe

```
C:\work>java Ex13_3
Server started: ServerSocket[addr=0.0.0.0/0.0.0.0,port=0,localport=8099]
Connection established: Socket[addr=/127.0.0.1,port=1067,localport=8099]
Client said: Client generated line 0
Client said: Client generated line 1
Client said: Client generated line 2
Client said: Client generated line 3
Client said: Client generated line 4
Client said: Client generated line 5
Client said: Client generated line 6
Client said: Client generated line 7
Client said: Client generated line 8
Client said: Client generated line 9
Closing connection...
Server shutdown...
```

Fig 13-1: **Output from examples.**

Establishing Connection

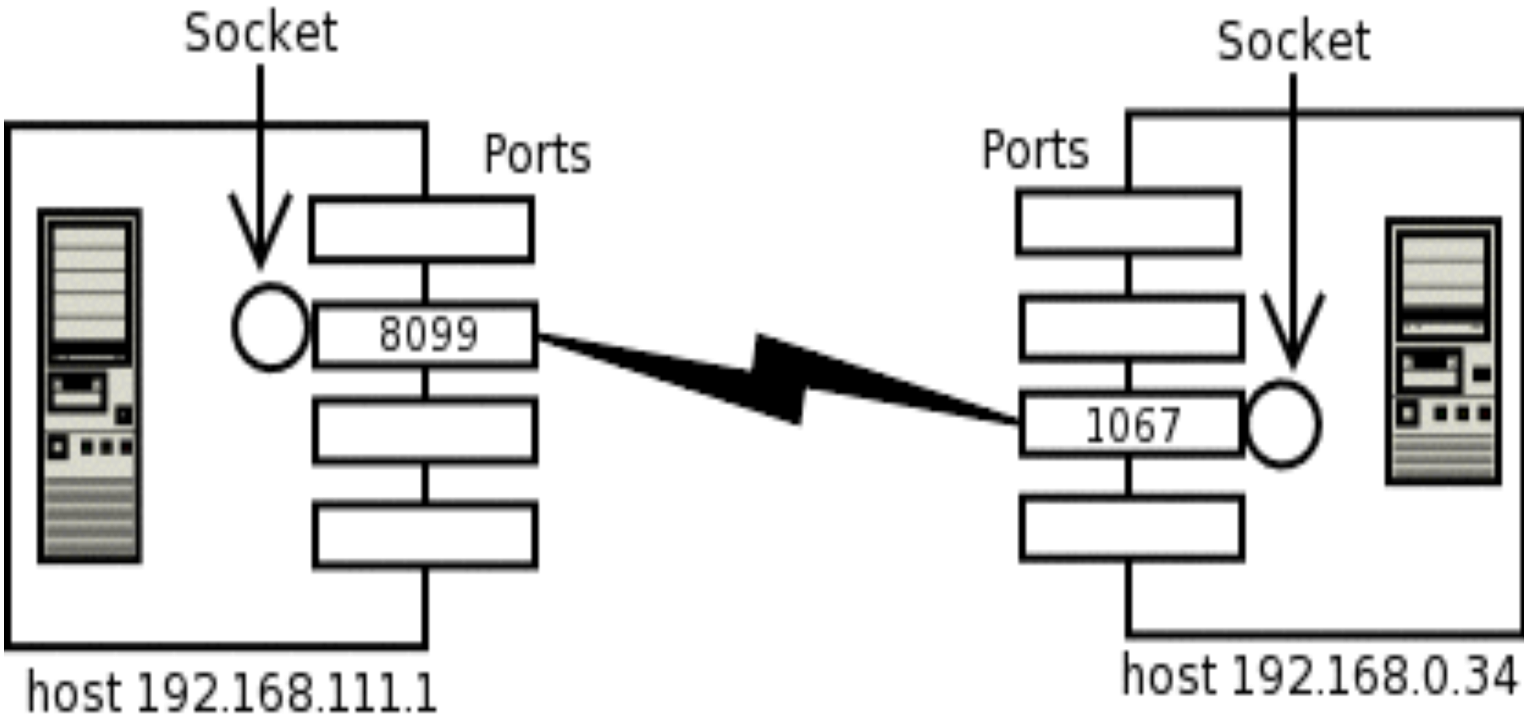


Fig 13-2: **An established connection with sockets on either end**

Multi-Threaded Server

Example 13-5: Multithreaded server

```
public class Ex13_5 {
  public static void main(String[] args) throws IOException {
    ServerSocket server = new ServerSocket(8099);
    System.out.println("Server started: " + server);
    try {
      while(true) {
        // Blocks until a connection occurs:
        Socket socket = server.accept();
        try {
          new ClientConnection(socket);
        } catch(IOException e) {
          // If it fails, close the socket,
          // otherwise the thread will close it:
          socket.close();
        }
      }
     } finally {
       server.close();
       System.out.println("Server shutdown...");
     }
  }
}
```

Multi-Threaded Server (cont.)

Example 13-6: Multithreaded server – Thread object

```
class ClientConnection extends Thread {
  private Socket thisSocket;
  private BufferedReader input;
  private PrintWriter output;

  public ClientConnection(Socket s) throws IOException {
     thisSocket = s;
     input = new BufferedReader(
               new InputStreamReader(thisSocket.getInputStream()));
     output = new PrintWriter(
                new BufferedWriter(new OutputStreamWriter(
                 thisSocket.getOutputStream())), true);

     start();
  }
```

Multi-Threaded Server (cont.)

Example 13-6: Multithreaded server – Thread object (continued)

```
  public void run() {
    try {
      while (true) {
        String s = input.readLine();
        if(s.equals("quit")) {
          break;
        }
        System.out.println("You said: " + s);
        output.println(s);
      }
      System.out.println("Closing connection...");
    } catch(IOException e) {
      System.err.println("IO Exception");
    } finally {
      try {
        thisSocket.close();
      } catch(IOException e) {
          System.err.println("Socket not closed");
      }
    }
  }
}
```

Reading from a URL

Example 13-7: Reading from a URL

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

public class  Reader {
  public static void main(String[] args) throws Exception {
     // Open the connection and get a Reader
     URL wl = new URL("http://www. .com/");
     BufferedReader in = new BufferedReader(
                              new InputStreamReader(wl.openStream()));
     // read from the URL
     String inputLine;
     while ((inputLine = in.readLine()) != null) {
       System.out.println(inputLine);
     }
     in.close();
  }
}
```

Summary

We covered

- Ports and sockets
- Clients and servers
- Writing a `ServerSocket` class
- Writing a client `Socket` class
- Writing a multi-threaded `ServerSocket`
- Reading from a URL