

Data Structures for Java

William H. Ford
William R. Topp



Chapter 2b Class Relationships

Bret Ford

© 2005, Prentice Hall

Static Binding



- Static binding associates a method with the class type of a reference variable.

Example:

```
emp.setName("Harrison, Pamela"),  
sEmp.setSalary(1500.0)
```



Overriding Methods

- When the superclass and the subclass have methods with the same signature, we say that the subclass method “overrides” the superclass method.

```
// emp = sEmp sets emp to point at sEmp ("Michael Morris").  
// the runtime system executes payrollCheck SalaryEmployee  
System.out.println(emp.payrollCheck());
```

Output:

Pay Morris, Mike (569-34-0382) \$1250.00



Polymorphism

- When the superclass and one or more subclasses define methods with the same signature, the runtime system executes the subclass method under the following conditions
 - A subclass object is assigned to a superclass reference variable.
 - The method call uses the superclass reference variable.

Polymorphism (concluded)



- Rather than using static binding which would associate the method with the superclass reference variable, the compiler directs the runtime system to determine the subclass type referenced by the variable and then calls the corresponding subclass method. This is dynamic binding since the association between reference variable and method is established at runtime.

Polymorphism Example



```
// declare a subclass object
HourlyEmployee hEmp =
    new HourlyEmployee("Holmes, Julie",
                       "837-68-2198",
                       12.00, 30);

// assign subclass object hEmp to superclass
// reference variable
emp = hEmp;
// create a pay check using polymorphism
System.out.println(emp.payrollCheck());
```

Output:

```
Pay Holmes, Julie (837-68-2198) $360.00
```

Upcasting



- Upcasting occurs when a subclass object reference is assigned to a superclass reference.
 - Superclass reference variable may call any public method in the superclass and any public method in the subclass for which polymorphism applies.

Downcasting



- A superclass reference variable may not be used to call a method defined exclusively in a subclass. The programmer must use a cast to change the reference type of the variable to that of the subclass.

```
SalaryEmployee sEmp =  
    new SalaryEmployee("Bonner, Al", "667-21-7128"  
        2500.0);  
  
Employee emp;  
  
Emp = sEmp;  
  
((SalaryEmployee)emp).setSalary(3000.0);
```



The instanceof Operator

- Sometimes the choice of a downcast cannot be made until runtime. The instanceof operator allows the determination of subclass type.
- The method `payIncrease()` provides a good example of using the instanceof operator. Its first argument can be either a `SalaryEmployee` or an `HourlyEmployee`.

```
// give employee a percentage pay increase of pct
public void payIncrease(Employee emp, double pct)
```



payIncrease()

```
public static void payIncrease(Employee emp,
                               double pct)
{
    // use instanceof to determine the
    // object type for emp if SalaryEmployee,
    // access and update salary
    if (emp instanceof SalaryEmployee)
        ((SalaryEmployee)emp).setSalary(
            (1.0 + pct) *
            ((SalaryEmployee)emp).getSalary());
    else
        ((HourlyEmployee)emp).setHourlyPay(
            (1.0 + pct) *
            ((HourlyEmployee)emp).getHourlyPay());
}
```



Abstract Classes

- Define an abstract method in a class by preceding the signature with the keyword `abstract` and replacing the method body by `;`. Place the keyword `abstract` in the class header.

```
abstract class ClassName
{
    // abstract class may contain data and concrete methods
    . . .
    // abstract class must contain at least one abstract method
    abstract public returnType methodName(<parameters>;
}
```



Abstract Classes (concluded)

- Each subclass of an abstract class must override all of the abstract methods in the superclass.
- A program cannot create an instance of an abstract class.
- Provides only resources for a subclass and method declarations that can be used with polymorphism.



The throw Statement

- An exception is an object that is created within a method at a point where an error condition occurs.
- Create the exception object in a throw statement that passes the object back through a chain of method calls to a block of code designed to catch the exception (an exception handler).

```
throw new ExceptionTypeException(errorMessage);
```



average()

```
// throws IllegalArgumentException
// object with the message
// "average(): invalid array"
public static double average(double[] arr)
{
    double sum = 0;

    // if array null or length 0, throw exception and exit
    if (arr == null || arr.length == 0)
        throw new IllegalArgumentException(
            "average(): Invalid array");
    // preconditions satisfied; compute and
    // return the average
    for (int i = 0; i < arr.length; i++)
        sum += arr[i];
    return sum/arr.length;
}
```

Handling an Exception



- The try statement is a block of code that may generate an exception.
- A catch block must immediately follow a try block and serves as the exception handler.

Try-Catch Statements:

```
try
{
    <program statements>
}
catch (ExceptionTypeException e)
{
    <display error message>
    < perform other tasks or exit the program>
}
```

Handling an Exception (concluded)



- If no exception occurs, processing occurs after the catch block.
- If a statement in a try block causes an exception, an immediate exit from the block occurs, and control passes to the catch block specified to handle the exception.

Try/Catch Block Example



```
public static void main(String[] args)
{
    // declare two array references;
    double[] arrA = {2.5, 5.0, 7.2, 8.1},
            arrB = null;

    try
    {
        // arrA is valid and average()
        // returns a value
        // arrB is not valid (null) and
        // average() throws an exception
        System.out.println("Average is " + average(arrA));
        System.out.println("Average is " + average(arrB));
    }
}
```

Try/Catch Block Example (concluded)



```
        catch (IllegalArgumentException e)
        {
            System.out.println(e);
            // exit the program
            System.exit(1);
        }
    }
```



Finally Clause

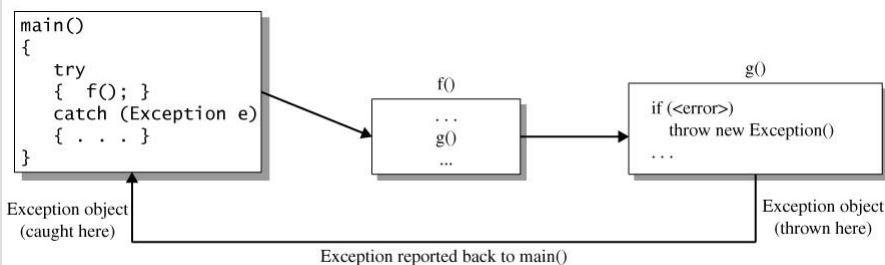
- The finally clause defines a block of code that always executes, no matter whether the try block executes successfully or causes an exception.
- Used typically to manage system resources, such as close files.

```
try
{ ... }
catch (ExceptionTypeException
e)
{ ... }
finally
{ ... }
```



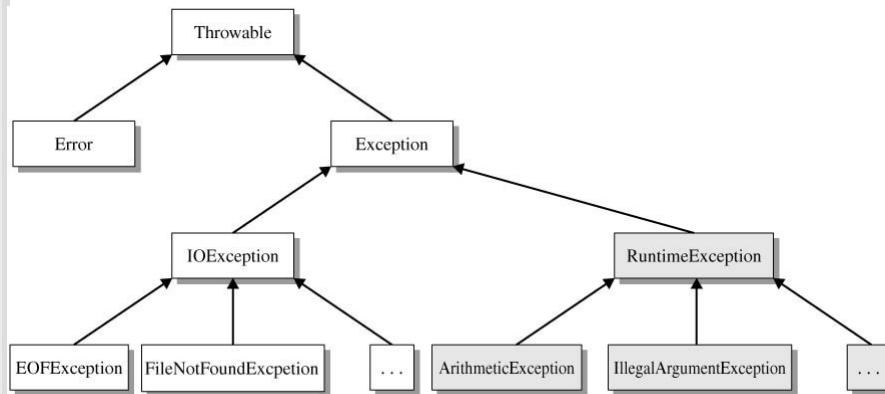
Error Propagation

- The exception handling mechanism may involve a chain of method calls with the exception occurring at some distant point in the chain.



Using a Java exception for an error in a chain of method calls.

Java Exception Hierarchy



Hierarchy tree for Java exceptions.

Standard Exceptions



- **ArithmeticException**
 - Thrown when an exceptional arithmetic condition has occurred. For example, an integer "divide by zero" throws an instance of this class.
- **IllegalArgumentException**
 - Thrown to indicate that a method has been passed an illegal or inappropriate argument
- **IndexOutOfBoundsException**
 - Thrown to indicate that an index of some sort (such as an array, a string, or an ArrayList) is out of range.

Standard Exceptions (concluded)



- **NullPointerException**
 - Thrown when an application attempts to use null in a case where an object is required
- **UnsupportedOperationException**
 - Thrown to indicate that the requested operation is not supported.

parseTime()



```
public static Time24 parseTime(String s)
{
    // tokens separated by space or colon character
    StringTokenizer stok =
        new StringTokenizer(s, " :");
    String timePeriod = null;
    int hour, minute;

    // get tokens and convert to hour and minute
    hour = Integer.parseInt(stok.nextToken());
    minute = Integer.parseInt(stok.nextToken());

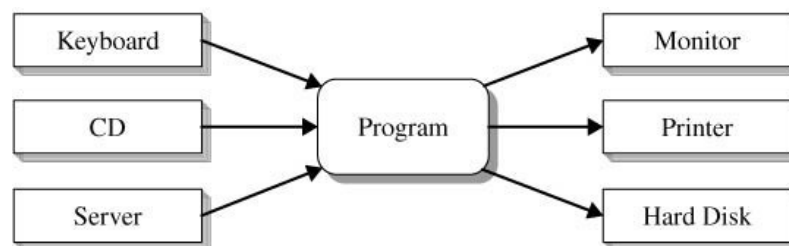
    // create a Time24 object
    // as the return value
    return new Time24(hour, minute);
}
```

Streams



- I/O streams carry data.
 - Text streams have character data such as an HTML file or a Java source file.
 - Binary streams have byte data that may represent a graphic or executable code, such as a Java .class file.
- A stream carries data from a source to a destination such as a monitor or hard disk.

Stream Sources and Destinations



Sources for input streams and destinations for output streams.

Types of Streams



- If data flows from a source into a program, it is called an input stream.
- If data flows from a program to a destination, it is called an output stream.
- The basic stream classes are defined in the package "java.io".

Console I/O



- Allows a program to input data from the keyboard and display output in a console window.
- The three console streams:
 - System.in for character input (standard input).
 - System.out for character output (standard output).
 - System.err character output (standard error).

Console Output Examples



- We are already familiar with console output.
 - Example:

```
System.out.print("Displayed in console window");  
System.err.println("Posting an error message");
```

File I/O



- A file is a collection of data that is stored in some medium such as a disk.
- A file that can be viewed as a sequence of characters partitioned into lines is a text file.
 - A text file can be listed on a console window or viewed with an editor.
- Nontext files are termed binary files.
 - Compressed data is stored in a binary file.

Text Input with Reader Streams



- Text input derived from the abstract class `java.io.Reader` which defines basic character input methods.
- The `java.io.FileReader` class inputs data from a disk file.

Example:

```
FileReader datIn =
    new
FileReader("testdata.dat");
```

Text Input with Reader Streams (concluded)

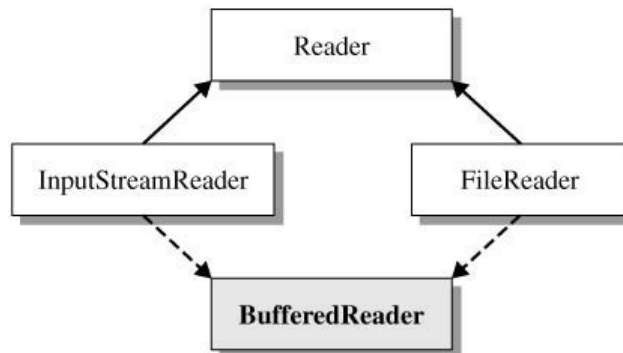


- `java.io.InputStream` is a bridge from byte streams to character streams. It reads bytes and decodes them into characters.
- `java.io.BufferedReader` reads text from a character-input stream, buffering characters to provide for the efficient reading of characters, arrays, and lines.

Example:

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

The Reader Hierarchy



The abstract Reader class with subclasses for buffered input from a file or the keyboard.

StringTokenizer



- `java.util.StringTokenizer` has methods that separate a string into tokens.

Example:

```
String str;
// tokens are blank and colon
StringTokenizer stok =
    new StringTokenizer(str, " :");

while (stok.hasMoreTokens())
    System.out.println(stok.nextToken());
```



parseTime()

```
public static Time24 parseTime(String s)
{
    // tokens separated by space or colon character
    StringTokenizer stok =
        new StringTokenizer(s, " :");
    String timePeriod = null;
    int hour, minute;

    // get tokens and convert to hour and minute
    hour = Integer.parseInt(stok.nextToken());
    minute = Integer.parseInt(stok.nextToken());

    // create a Time24 object as the return value
    return new Time24(hour, minute);
}
```



Text Output with Writer Streams

- The abstract `java.io.Writer` class defines basic operations for inserting a character or arrays of characters into a stream.
- The class `java.io.FileWriter` creates objects that are attached to a file on disk.

Example:

```
FileWriter dataOut = new FileWriter("output.dat");
```



Output with Print Statements

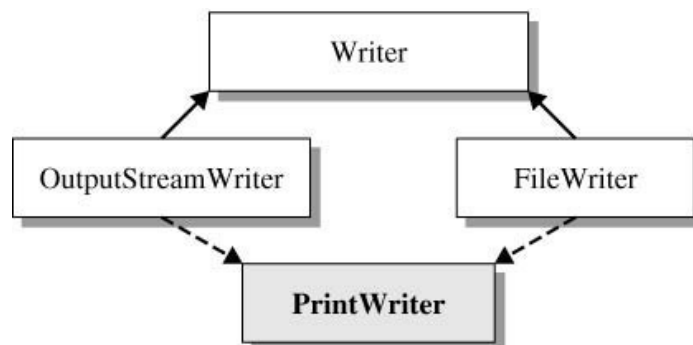
- The class `java.io.PrintWriter` wraps around a `Writer` object and outputs formatted data.

Example:

```
PrintWriter pw = new PrintWriter(  
    new FileWriter("data.out"));  
int n = 5;  
  
pw.println("n = " + n);
```



Writer Inheritance Hierarchy



Writer inheritance hierarchy with physical character stream classes and filter class `PrintWriter`.

Controlling the Output Stream



- The method `flush()` directs the I/O system to copy all buffered data to the output stream.
- The method `close()` closes the stream and releases all resources associated with the stream.

The Scanner Class



- A Scanner object partitions text from an input stream into tokens by means of its "next" methods.
- Declare a Scanner object as follows:

```
Scanner keyIn = new Scanner(System.in);  
Scanner fileIn = new Scanner(  
    new FileReader("demo.dat"));
```

Scanner Methods



```
Shortcut to cmd.exe
C:\> 17 deposit 450.75 false A
```

- `String line = sc.nextLine(); // whole line`
- `int i = sc.nextInt(); // i = 17`
- `String str = sc.next(); // str = "deposit"`
- `double x = sc.nextDouble(); // x = 450.75`
- `boolean b = sc.nextBoolean(); // b = false`
- `char ch = sc.next().charAt(0); // ch = 'A'`

Testing for Scanner Tokens



```
// loop reads tokens in the line
while (sc.hasNext())
{
    token = sc.next();
    System.out.println("In loop next token = " + token);
}
```

Output:

```
In loop next token = 17
In loop next token = deposit
In loop next token = 450.75
In loop next token = false
In loop next token = A
```



File Input using Scanner

A sports team creates the file "attendance.dat" to store attendance data for home games during the season. The example uses the Scanner object dataIn and a loop to read the sequence of attendance values from the file and determine the total attendance. The condition hasNextInt() returns false when all of the data has been read from the file.

```
// create an Scanner object attached to the file "attendance.dat"
Scanner dataIn = new Scanner(new FileReader("attendance.dat"));
int gameAtt, totalAtt = 0;

// the loop reads the integer game attendance until end-of-file
while(dataIn.hasNextInt())           // loop iteration condition
{
    gameAtt = dataIn.nextInt(); // input next game attendance
    totalAtt += gameAtt;        // add to the total
}
```



Scanner Class API

class SCANNER	java.util
Constructors	
	Scanner ((InputStream source) Creates a Scanner object that produces values read from the specified input stream (Typically standard input System.in that denotes the keyboard)
	Scanner (Readable source) Creates a Scanner object that produces values read from the specified input stream (Typically a FileReader that denotes a file)

Scanner Class API continued



Methods	
void	close() Close the scanner.
boolean	hasNext() Returns true if the scanner has another token in the input stream
boolean	hasNextBoolean() Returns true if the next token in the input stream can be interpreted as a boolean value
boolean	hasNextDouble() Returns true if the next token in the input stream can be interpreted as a double value
boolean	hasNextInt() Returns true if the next token in the input stream can be interpreted as an int value

Scanner Class API (concluded)



String	next() Finds and returns the next complete token in the input stream as a String.
boolean	nextBoolean() Scans the next token in the input stream into a boolean value and returns that value.
double	nextDouble() Scans the next token in the input stream into a double value and returns that value.
int	nextInt () Scans the next token in the input stream into an int value and returns that value.

Program 2.1



```
import java.util.Scanner;
import java.io.*;
import java.text.DecimalFormat;

public class Program2_1 {
    public static void main(String[] args)
    {
        final double SALESTAX = 0.05;

        // input streams for the keyboard and a file
        Scanner fileIn = null;
        // input variables and pricing information
        String product;
        int quantity;
        double unitPrice, quantityPrice, tax,
            totalPrice;
        char taxStatus;
```

Program 2.1 (continued)



```
        // create formatted strings for
        // aligning output
        DecimalFormat fmtA = new DecimalFormat("#"),
            fmtB = new DecimalFormat("$#.00");

        // open the file; catch exception
        // if file not found
        // use regular expression as delimiter
        try
        {
            fileIn =
                new Scanner(new FileReader("food.dat"));
            fileIn.useDelimiter("[\\t\\n\\r]+");
        }
```

Program 2.1 (continued)



```
catch (IOException ioe)
{
    System.err.println("Cannot open file " +
                       "'food.dat'");
    System.exit(1);
}

// header for listing output
System.out.println("Product" +
                  align("Quantity", 16) +
                  align("Price", 10) +
                  align("Total", 12));
```

Program 2.1 (continued)



```
// read to end of file; break
// when no more tokens
while(fileIn.hasNext())
{
    // input product/purchase input fields
    product = fileIn.next();
    quantity = fileIn.nextInt();
    unitPrice = fileIn.nextDouble();
    taxStatus = fileIn.next().charAt(0);

    // calculations and output
    quantityPrice = unitPrice * quantity;
    tax = (taxStatus == 'Y') ?
          quantityPrice *
          SALESTAX : 0.0;
    totalPrice = quantityPrice + tax;
```

Program 2.1 (concluded)



```
        System.out.println(product +
            align("", 15-product.length()) +
            align(fmtA.format(quantity), 6) +
            align(fmtB.format(unitPrice), 13) +
            align(fmtB.format(totalPrice), 12) +
            ((taxStatus == 'Y') ? " *" : ""));
    }
}
// aligns string right justified in a field of width n
public static String align(String str, int n) {
    String alignStr = "";
    for (int i = 1; i < n - str.length(); i++)
        alignStr += " ";
    alignStr += str;
    return alignStr;
}
}
```

Program 2.1 Run



```
Input file ('food.dat')
Soda    3 2.69 Y
Eggs    2 2.89 N
Bread   3 2.49 N
Grapefruit  8 0.45 N
Batteries 10 1.15 Y
Bakery  1 14.75 N
```

Run:

Product	Quantity	Price	Total
Fruit Punch	4	\$2.69	\$11.30 *
Eggs	2	\$2.89	\$5.78
Rye Bread	3	\$2.49	\$7.47
Grapefruit	8	\$.45	\$3.60
AA Batteries	10	\$1.15	\$12.08 *
Ice Cream	1	\$3.75	\$3.75