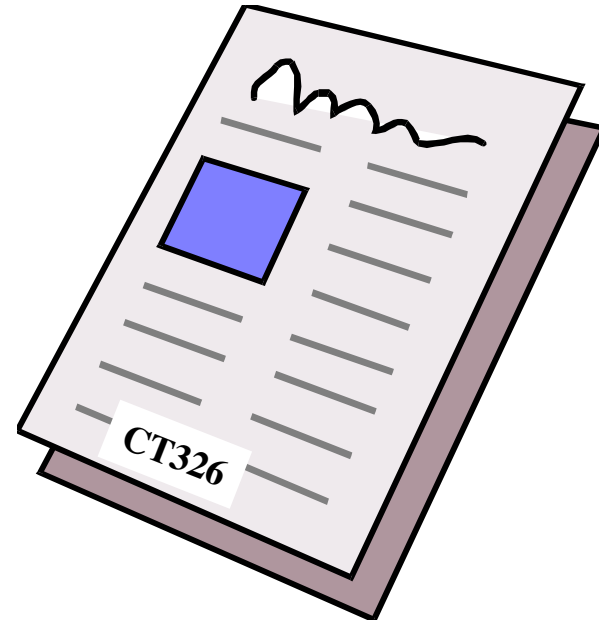


# Programming III

---

**IT Dept  
NUI Galway**

*Dr. Des Chambers  
des.chambers@nuigalway.ie  
Room IT408*



# JAVA

---

- Textbook for this course:
  - » Java – How to Program by Deitel & Deitel
  - » Available in University Bookshop
  - » Lab Assignments every week

# JAVA

---

- » Quick Revision of Java Programming
  - Java uses a class to represent objects.
  - An object is a thing upon which your application performs different operations.
  - A class contains members - these may be:
    - Information (or data) often called class variables.
    - Functions (methods) that operate on the data.
  - Each class has a unique name.
  - To create an instance of a class variable, you must use the *new* operator.

# Using Overloaded Constructors

- Overloaded constructors
  - Methods (in same class) may have same name
  - Must have different parameter lists

```

1 // Fig. 8.6: Time2.java
2 // Time2 class definition with overloaded constructors.
3 package com.deitel.jhttp4.ch08;
4
5 // Java core packages
6 import java.text.DecimalFormat;
7
8 public class Time2 extends Object {
9     private int hour;      // 0 - 23
10    private int minute;    // 0 - 59
11    private int second;    // 0 - 59
12
13    // Time2 constructor initializes each instance variable
14    // to zero. Ensures that Time object starts in a
15    // consistent state.
16    public Time2()
17    {
18        setTime( 0, 0, 0 );
19    }
20
21    // Time2 constructor: hour supplied, minute and second
22    // defaulted to 0
23    public Time2( int h )
24    {
25        setTime( h, 0, 0 );
26    }
27
28    // Time2 constructor: hour and minute supplied, second
29    // defaulted to 0
30    public Time2( int h, int m )
31    {
32        setTime( h, m, 0 );
33    }
34

```

Default constructor has no arguments

Overloaded constructor  
has one **int** argument

Second overloaded constructor has  
two **int** arguments

## Time2.java

Lines 16-19

Default constructor has  
no arguments

Lines 23-26

Overloaded constructor  
has one **int** argument

Lines 30-33

Second overloaded  
constructor has two **int**  
arguments

```

35 // Time2 constructor: hour, minute and second supplied
36 public Time2( int h, int m, int s )
37 {
38     setTime( h, m, s );
39 }
40
41 // Time2 constructor: another Time2 object supplied
42 public Time2( Time2 time )
43 {
44     setTime( time.hour, time.minute, time.second );
45 }
46
47 // Set a new time value using universal time
48 // validity checks on data. Set invalid values to 0
49 public void setTime( int h, int m, int s )
50 {
51     hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
52     minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
53     second = ( ( s >= 0 && s < 60 ) ? s : 0 );
54 }
55
56 // convert to String in universal-time format
57 public String toUniversalString()
58 {
59     DecimalFormat twoDigits = new DecimalFormat( "00" );
60
61     return twoDigits.format( hour ) + ":" +
62         twoDigits.format( minute ) + ":" +
63         twoDigits.format( second );
64 }
65
66 // convert to String in standard-time format
67 public String toString()
68 {
69     DecimalFormat twoDigits = new DecimalFormat( "00" );

```

Third overloaded constructor has three **int** arguments

**Time2.java**

Lines 36-39

Third overloaded constructor has three **int** arguments

Fourth overloaded constructor has **Time2** argument

Lines 42-45

Fourth overloaded constructor has **Time2** argument

```
70
71     return ( (hour == 12 || hour == 0) ? 12 : hour % 12 ) +
72         ":" + twoDigits.format( minute ) +
73         ":" + twoDigits.format( second ) +
74         ( hour < 12 ? " AM" : " PM" );
75 }
76
77 } // end class Time2
```

**Time2.java**

```

1  // Fig. 8.7: TimeTest4.java
2  // Using overloaded constructors
3
4  // Java extension packages
5  import javax.swing.*;
6
7  // Deitel packages
8  import com.deitel.jhttp4.ch08.Time2;
9
10 public class TimeTest4 {
11
12     // test constructors of class Time2
13     public static void main( String args[] )
14     {
15         Time2 t1, t2, t3, t4, t5, t6;
16
17         t1 = new Time2();
18         t2 = new Time2( 2 );
19         t3 = new Time2( 21, 34 );
20         t4 = new Time2( 12, 25, 42 );
21         t5 = new Time2( 27, 74, 99 );
22         t6 = new Time2( t4 );
23
24         String output = "Constructed with: " +
25             "\nt1: all arguments defaulted" +
26             "\n      " + t1.toUniversalString() +
27             "\n      " + t1.toString();
28
29         output += "\nt2: hour specified; minute and " +
30             "second defaulted" +
31             "\n      " + t2.toUniversalString() +
32             "\n      " + t2.toString();
33

```

Declare six references to **Time2** objects

Instantiate each **Time2** reference  
using a different constructor

**TimeTest4.java**

Line 15

Declare six references to  
**Time2** objects

Lines 17-22

Instantiate each **Time2**  
reference using a  
different constructor

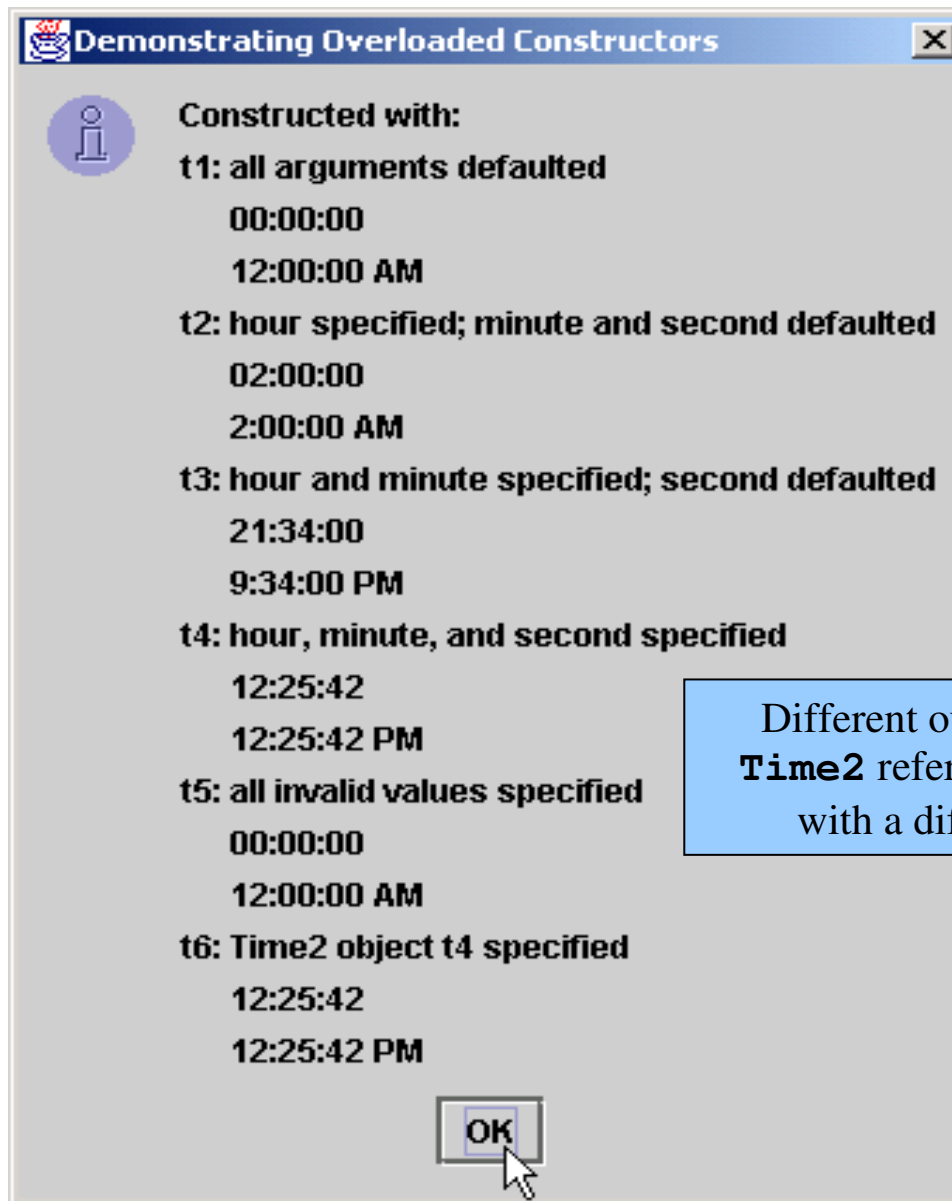


```

34     output += "\nt3: hour and minute specified; " +
35             "second defaulted" +
36             "\n        " + t3.toUniversalString() +
37             "\n        " + t3.toString();
38
39     output += "\nt4: hour, minute, and second specified" +
40             "\n        " + t4.toUniversalString() +
41             "\n        " + t4.toString();
42
43     output += "\nt5: all invalid values specified" +
44             "\n        " + t5.toUniversalString() +
45             "\n        " + t5.toString();
46
47     output += "\nt6: Time2 object t4 specified" +
48             "\n        " + t6.toUniversalString() +
49             "\n        " + t6.toString();
50
51     JOptionPane.showMessageDialog( null, output,
52     "Demonstrating Overloaded Constructors",
53     JOptionPane.INFORMATION_MESSAGE );
54
55     System.exit( 0 );
56 }
57
58 } // end class TimeTest4

```

**TimeTest4.java**



**TimeTest4.java**

Different outputs,  
because each **Time2**  
reference was instantiated  
with a different  
constructor

Different outputs, because each  
**Time2** reference was instantiated  
with a different constructor

# JAVA

---

- Class Inheritance

- When your applications use inheritance, you use a *super* class to derive a new class:
  - The new class inherits the *super* class members.
- To initialise class members for an extended class (called a subclass), application invokes the *super* class and subclass constructors.
- Use the *this* and *super* keywords to resolve.
- There are three types of members:
  - **public, private and protected**

# JAVA

---

## » Access Level Specifiers

	Class	Package	Subclass	World
private	Y	N	N	N
no specifier	Y	Y	N	N
protected	Y	Y	Y	N
public	Y	Y	Y	Y

# Case Study: A Payroll System Using Polymorphism

- Abstract methods and polymorphism
  - Abstract superclass **Employee**
    - Method **earnings** applies to all employees
    - Person's earnings dependent on type of **Employee**
  - Concrete **Employee** subclasses declared **final**
    - **Boss**
    - **CommissionWorker**
    - **PieceWorker**
    - **HourlyWorker**
  - Chapter 10 of Deitels Book covers a similar example and has the code on the CD.

```

1  // Fig. 9.16: Employee.java
2  // Abstract base class Employee.
3
4  public abstract class Employee {
5      private String firstName;
6      private String lastName;
7
8      // constructor
9      public Employee( String first, String last )
10     {
11         firstName = first;
12         lastName = last;
13     }
14
15     // get first name
16     public String getFirstName()
17     {
18         return firstName;
19     }
20
21     // get last name
22     public String getLastName()
23     {
24         return lastName;
25     }
26
27     public String toString()
28     {
29         return firstName + ' ' + lastName;
30     }
31

```

**abstract** class cannot be instantiated

**abstract** class can have instance data and non**abstract** methods for subclasses

**abstract** class can have constructors for subclasses to initialize inherited data

Employee.java


**abstract** class cannot be instantiated

Lines 5-6 and 16-30  
**abstract** class can have instance data and non**abstract** methods for subclasses

Lines 9-13  
**abstract** class can have constructors for subclasses to initialize inherited data

```
32    // Abstract method that must be implemented for each
33    // derived class of Employee from which objects
34    // are instantiated.
35    public abstract double earnings();
36
37 } // end class Employee
```

Subclasses must implement  
**abstract** method



**Employee.java**

Line 35  
Subclasses must  
implement **abstract**  
method

```

1  // Fig. 9.17: Boss.java
2  // Boss class derived from Employee.
3
4  public final class Boss extends Employee {
5      private double weeklySalary;
6
7      // constructor for class Boss
8      public Boss( String first, String last, double salary )
9      {
10         super( first, last ); // call superclass constructor
11         setWeeklySalary( salary );
12     }
13
14     // set Boss's salary
15     public void setWeeklySalary( double salary )
16     {
17         weeklySalary = ( salary > 0 ? salary : 0 );
18     }
19
20     // get Boss's pay
21     public double earnings()
22     {
23         return weeklySalary;
24     }
25
26     // get String representation of Boss's name
27     public String toString()
28     {
29         return "Boss: " + super.toString();
30     }
31
32 } // end class Boss

```

**Boss** is an **Employee** subclass

**Boss** inherits **Employee's** public methods (except for constructor)

Explicit call to **Employee** constructor using **super**

Required to implement **Employee's** method **earnings** (polymorphism)

**Boss** is an **Employee** subclass

Line 4  
**Boss** inherits **Employee's** public methods (except for constructor)

Explicit call to **Employee** constructor using **super**

Lines 21-24  
Required to implement **Employee's** method **earnings** (polymorphism)



**CommissionWorker** is an  
**Employee** subclass

```
1  // Fig. 9.18: CommissionWorker.java
2  // CommissionWorker class derived from Employee
3
4  public final class CommissionWorker extends Employee {
5      private double salary;           // base salary per week
6      private double commission;      // amount per item sold
7      private int quantity;           // total items sold for week
8
9      // constructor for class CommissionWorker
10     public CommissionWorker( String first, String last,
11         double salary, double commission, int quantity )
12     {
13         super( first, last ); // call superclass constructor
14         setSalary( salary );
15         setCommission( commission );
16         setQuantity( quantity );
17     }
18
19     // set CommissionWorker's weekly base salary
20     public void setSalary( double weeklySalary )
21     {
22         salary = ( weeklySalary > 0 ? weeklySalary : 0 );
23     }
24
25     // set CommissionWorker's commission
26     public void setCommission( double itemCommission )
27     {
28         commission = ( itemCommission > 0 ? itemCommission : 0 );
29     }
30
```

Explicit call to **Employee**  
constructor using **super**

**CommissionWorker.java**

Line 4  
**CommissionWorker**  
is an **Employee**  
subclass

Line 13  
Explicit call to  
**Employee** constructor  
using **super**

```

31 // set CommissionWorker's quantity sold
32 public void setQuantity( int t
33 {
34     quantity = ( totalSold > 0
35 }
36
37 // determine CommissionWorker's earnings
38 public double earnings()
39 {
40     return salary + commission * quantity;
41 }
42
43 // get String representation of CommissionWorker's name
44 public String toString()
45 {
46     return "Commission worker: " + super.toString();
47 }
48
49 } // end class CommissionWorker

```

Required to implement **Employee's** method **earnings**; this implementation differs from that in **Boss**

**CommissionWorker.java**

Lines 38-41  
Required to implement **Employee's** method **earnings**; this implementation differs from that in **Boss**

**PieceWorker** is an **Employee** subclass

```
1  // Fig. 9.19: PieceWorker.java
2  // PieceWorker class derived from Employee
3
4  public final class PieceWorker extends Employee {
5      private double wagePerPiece; // wage per piece output
6      private int quantity;        // output for week
7
8      // constructor for class PieceWorker
9      public PieceWorker( String first, String last,
10         double wage, int numberOfItems )
11      {
12          super( first, last ); // call superclass constructor
13          setWage( wage );
14          setQuantity( numberOfItems );
15      }
16
17      // set PieceWorker's wage
18      public void setWage( double wage )
19      {
20          wagePerPiece = ( wage > 0 ? wage : 0 );
21      }
22
23      // set number of items output
24      public void setQuantity( int numberOfItems )
25      {
26          quantity = ( numberOfItems > 0 ? numberOfItems : 0 );
27      }
28
29      // determine PieceWorker's earnings
30      public double earnings()
31      {
32          return quantity * wagePerPiece;
33      }
34
```

Explicit call to **Employee** constructor using **super**

Implementation of **Employee**'s method **earnings**; differs from that of **Boss** and **CommissionWorker**

**PieceWorker.java**

Line 4

**PieceWorker** is an **Employee** subclass

Line 12

Explicit call to **Employee** constructor using **super**

Lines 30-33

Implementation of **Employee**'s method **earnings**; differs from that of **Boss** and **CommissionWorker**

```
35     public String toString()
36     {
37         return "Piece worker: " + super.toString();
38     }
39
40 } // end class PieceWorker
```

**PieceWorker.java**

HourlyWorker is an  
Employee subclass

```
1 // Fig. 9.20: HourlyWorker.java
2 // Definition of class HourlyWorker
3
4 public final class HourlyWorker extends Employee {
5     private double wage;    // wage per hour
6     private double hours;   // hours worked for week
7
8     // constructor for class HourlyWorker
9     public HourlyWorker( String first, String last,
10         double wagePerHour, double hoursWorked )
11     {
12         super( first, last );    // call superclass constructor
13         setWage( wagePerHour );
14         setHours( hoursWorked );
15     }
16
17     // Set the wage
18     public void setWage( double wagePerHour )
19     {
20         wage = ( wagePerHour > 0 ? wagePerHour : 0 );
21     }
22
23     // Set the hours worked
24     public void setHours( double hoursWorked )
25     {
26         hours = ( hoursWorked >= 0 && hoursWorked < 168 ?
27             hoursWorked : 0 );
28     }
29
30     // Get the HourlyWorker's pay
31     public double earnings() { return wage * hours; }
32
```

Explicit call to **Employee**  
constructor using **super**

Implementation of **Employee's** method  
**earnings**; differs from that of other  
**Employee** subclasses

HourlyWorker.java

Line 4

**PieceWorker** is an  
**Employee** subclass

Line 12

Explicit call to  
**Employee** constructor  
using **super**

Line 31

Implementation of  
**earnings** method  
differs from  
**Employee**  
subclasses

```
33     public String toString()  
34     {  
35         return "Hourly worker: " + super.toString();  
36     }  
37  
38 } // end class HourlyWorker
```

**HourlyWorker.java**

```

1  // Fig. 9.21: Test.java
2  // Driver for Employee hierarchy
3
4  // Java core packages
5  import java.text.DecimalFormat;
6
7  // Java extension packages
8  import javax.swing.JOptionPane;
9
10 public class Test {
11
12     // test Employee hierarchy
13     public static void main( String args[] )
14     {
15         Employee employee; // superclass reference
16         String output = "";
17
18         Boss boss = new Boss( "John", "Smith", 800.0 );
19
20         CommissionWorker commissionWorker =
21             new CommissionWorker(
22                 "Sue", "Jones", 400.0, 3.0, 150 );
23
24         PieceWorker pieceWorker =
25             new PieceWorker( "Bob", "Lewis", 2.5, 200 );
26
27         HourlyWorker hourlyWorker =
28             new HourlyWorker( "Karen", "Price", 13.75, 40 );
29
30         DecimalFormat precision2 = new DecimalFormat( "0.00" );
31

```

**Test.java**

**Test** cannot instantiate **Employee** but  
can reference one

Line 15

**Test** cannot instantiate  
**Employee** but can  
reference one

Instantiate one instance each of  
**Employee** subclasses

one instance  
each of **Employee**  
subclasses

Use **Employee** to reference **Boss**

```
32 // Employee reference to a Boss
33 employee = boss;
```

Test.java

Method **employee.earnings**  
dynamically binds to method  
**boss.earnings**

```
35 output += employee.toString() + " earned $" +
36     precision2.format( employee.earnings() ) + "\n" +
37     boss.toString() + " earned $" +
38     precision2.format( boss.earnings() ) + "\n";
```

```
40 // Employee reference to a CommissionWorker
41 employee = commissionWorker;
```

Do same for **CommissionWorker**  
and **PieceWorker**

```
42
43 output += employee.toString() + " earned $" +
44     precision2.format( employee.earnings() ) + "\n" +
45     commissionWorker.toString() + " earned $" +
46     precision2.format(
47     commissionWorker.earnings() ) + "\n";
```

```
49 // Employee reference to a PieceWorker
50 employee = pieceWorker;
```

```
51
52 output += employee.toString() + " earned $" +
53     precision2.format( employee.earnings() ) + "\n" +
54     pieceWorker.toString() + " earned $" +
55     precision2.format( pieceWorker.earnings() ) + "\n";
```

Line 36  
Method  
**employee.earnings**  
to  
**boss.earnings**

Lines 41-55  
Do same for  
**CommissionWorker**  
and **PieceWorker**



```

57 // Employee reference to an HourlyWorker
58 employee = hourlyWorker;
59
60 output += employee.toString() + " earned $" +
61 precision2.format( employee.earnings() ) + "\n" +
62     hourlyWorker.toString() + " earned $" +
63     precision2.format( hourlyWorker.earnings() ) + "\n";
64
65 JOptionPane.showMessageDialog( null, output,
66     "Demonstrating Polymorphism",
67     JOptionPane.INFORMATION_MESSAGE );
68
69 System.exit( 0 );
70 }
71
72 } // end class Test

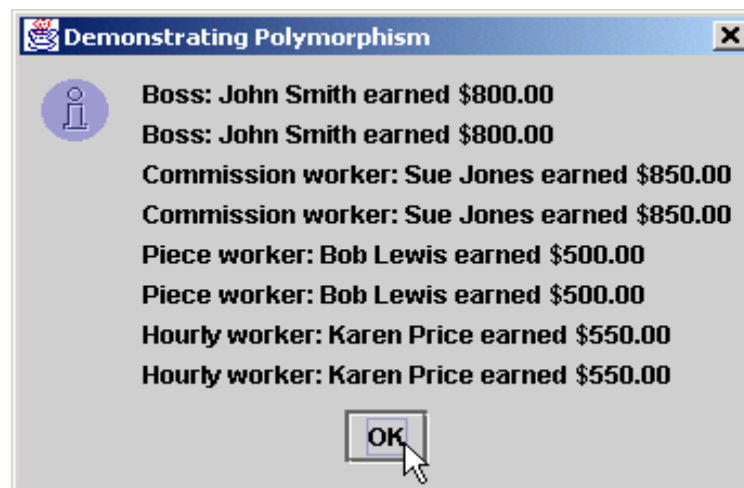
```

**Test.java**

Lines 58-63

Repeat for  
**HourlyWorker**

Repeat for **HourlyWorker**



# Lab Assignment One

---

# JAVA

---

- Solving Common Coding Problems
- Problem: The compiler complains that it can't find a class.
  - » Make sure you've imported the class or its package.
  - » Unset the CLASSPATH environment variable, if it's set.
  - » Make sure you're spelling the class name exactly the same way as it is declared. Case matters!
  - » Also, some programmers use different names for the class name from the .java filename. Make sure you're using the class name and not the filename. In fact, make the names the same and you won't run into this problem for this reason.

# JAVA

---

- Problem: The interpreter says it can't find one of my classes.
  - » Make sure you specified the class name--not the class file name--to the interpreter.
  - » Unset the CLASSPATH environment variable, if it's set.
  - » If your classes are in packages, make sure that they appear in the correct subdirectory.
  - » Make sure you're invoking the interpreter from the directory in which the .class file is located.

# JAVA

---

- Problem: My program doesn't work! What's wrong with it?
  - » Did you forget to use break after each case statement in a switch statement?
  - » Did you use the assignment operator = when you really wanted to use the comparison operator ==?
  - » Are the termination conditions on your loops correct? Make sure you're not terminating loops one iteration too early or too late. That is, make sure you are using < or <= and > or >= as appropriate for your situation.
  - » Remember that array indices begin at 0, so iterating over an array looks like this:

```
for (int i = 0; i < array.length; i++)  
    ...
```

# JAVA

---

- Are you comparing floating-point numbers using ==?
  - » Remember that floats are approximations of the real thing.
  - » The greater than and less than (> and <) operators are more appropriate when conditional logic is performed on floating-point numbers.
- Are you having trouble with encapsulation, inheritance, or other object-oriented programming and design concepts?

# JAVA

---

- Make sure that blocks of statements are enclosed in curly brackets { }.
  - » The following code looks right because of indentation, but it doesn't do what the indents imply because the brackets are missing:

```
for (int i = 0; i < arrayOfInts.length; i++)  
    arrayOfInts[i] = i;  
    System.out.println("[i] = " + arrayOfInts[i]);
```

# JAVA

---

- Are you using the correct conditional operator? Make sure you understand && and || and are using them appropriately.
- Do you use the negation operator ! a lot? Try to express conditions without it. Doing so is less confusing and error-prone.
- Are you using a do-while? If so, do you know that a do-while executes at least once, but a similar while loop may not be executed at all?



# JAVA

---

- Are you trying to change the value of an argument from a method? Simple value arguments (like int) are passed by value and can't be changed in a method.
- Did you inadvertently add an extra semicolon (;), thereby terminating a statement prematurely? Notice the extra semicolon at the end of this for statement:

```
for (int i = 0; i < arrayOfInts.length; i++) ;  
    arrayOfInts[i] = i;
```

# Exception Handling Should Be Used

- Uses of exception handling
  - When method cannot complete its task
  - Process exceptions from program components
  - Handle exceptions in a uniform manner in large projects

# Other Error-Handling Techniques

- Using no error-handling
  - Not for mission critical applications
- Exit application on error
  - Program must return resources

# Basics of Java Exception Handling

- A method detects an error and throws an exception
  - Exception handler processes the error
    - The error is considered caught and handled in this model
- Code that could generate errors put in **try** blocks
  - Code for error handling enclosed in a **catch** block
  - The **finally** always executes with or without an error
- Keyword **throws** tells exceptions of a method
- Termination model of exception handling
  - The block in which the exception occurs expires

# try Blocks

- The **try** block structure

```
try {  
    statements that may throw an exception  
}  
catch ( ExceptionType exceptionReference ) {  
    statements to process an exception  
}
```

- A **try** followed by any number of **catch** blocks

# Throwing an Exception

- The **throw** statement
  - Indicates an exception has occurred
  - Operand any class derived from **Throwable**
- Subclasses of **Throwable**
  - Class **Exception**
    - Problems that should be caught
  - Class **Error**
    - Serious exception should not be caught
- Control moves from **try** block to catch **block**

# Catching an Exception

- Handler catches exception
  - Executes code in **catch** block
  - Should only catch **Exceptions**
- Program terminates if no appropriate handler
- Single **catch** can handle multiple exceptions
- Many ways to write exception handlers
- Rethrow exception if **catch** cannot handle it

# throws Clause

- Lists the exceptions thrown by a method

```
int functionName ( parameterList )  
    throws ExceptionType1, ExceptionType2,...  
{  
    // method body  
}
```

- **RuntimeExceptions** occur during execution
  - **ArrayIndexOutOfBoundsException**
  - **NullPointerException**
- Declare exceptions a method throws



# Finalizers, and Exception Handling

- Throw exception if constructor causes error
- **Finalize** called when object garbage collected
- Inheritance of exception classes
  - Allows polymorphic processing of related exceptions

# finally Block

- Resource leak
  - Caused when resources are not released by a program
- The **finally** block
  - Appears after **catch** blocks
  - Always executes
  - Use to release resources

```

1  // Fig. 14.9: UsingExceptions.java
2  // Demonstration of the try-catch-finally
3  // exception handling mechanism.
4  public class UsingExceptions {
5
6      // execute application
7      public static void main( String args[] )
8      {
9          // call method throwException
10         try {
11             throwException();
12         }
13
14         // catch Exceptions thrown by method throwException
15         catch ( Exception exception )
16         {
17             System.err.println( "Exception handled in main" );
18         }
19
20         doesNotThrowException();
21     }
22
23     // demonstrate try/catch/finally
24     public static void throwException() throws Exception
25     {
26         // throw an exception and immediately catch it
27         try {
28             System.out.println( "Method throwException" );
29             throw new Exception(); // generate exception
30         }
31

```

Method **main**  
immediately enters  
**try** block

Calls method  
**throwException**

Handle exception  
thrown by  
**throwException**

Call method  
**doesNotThrow-**  
**Exception**

Method throws new  
**Exception**

```

32     // catch exception thrown in try block
33     catch ( Exception exception )
34     {
35         System.err.println(
36             "Exception handled in method throwException" );
37         throw exception; // rethrow for further processing
38
39         // any code here would not be reached
40     }
41
42     // this block executes regardless of what occurs in
43     // try/catch
44     finally {
45         System.err.println(
46             "Finally executed in throwException" );
47     }
48
49     // any code here would not be reached
50 }
51
52 // demonstrate finally when no exception occurs
53 public static void doesNotThrowException()
54 {
55     // try block does not throw an exception
56     try {
57         System.out.println( "Method doesNotThrowException" );
58     }
59
60     // catch does not execute, because no exception thrown
61     catch( Exception exception )
62     {
63         System.err.println( exception.toString() );
64     }
65

```

Catch **Exception**

Rethrow **Exception**

The **finally** block  
executes, even though  
**Exception** thrown


Skip **catch** block  
since no **Exception**  
thrown

```

66      // this block executes regardless of what occurs in
67      // try/catch
68      finally {
69          System.err.println(
70              "Finally executed in doesNotThrowException" );
71      }
72
73      System.out.println(
74          "End of method doesNotThrowException" );
75  }
76
77  } // end class UsingExceptions

```

The **finally** block  
always executes



```

Method throwException
Exception handled in method throwException
Finally executed in throwException
Exception handled in main
Method doesNotThrowException
Finally executed in doesNotThrowException
End of method doesNotThrowException !

```

## 14.14 Using `printStackTrace` and `getMessage`

- Method `printStackTrace`
  - Prints the method call stack
- `Throwable` class
  - Method `getMessage` retrieves `informationString`

```

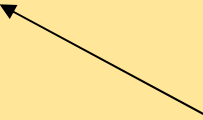
1  // Fig. 14.11: UsingExceptions.java
2  // Demonstrating the getMessage and printStackTrace
3  // methods inherited into all exception classes.
4  public class UsingExceptions {
5
6      // execute application
7      public static void main( String args[] )
8      {
9          // call method1
10         try {
11             method1();
12         }
13
14         // catch Exceptions thrown from method1
15         catch ( Exception exception ) {
16             System.err.println( exception.getMessage() + "\n" );
17             exception.printStackTrace();
18         }
19     }
20
21     // call method2; throw exceptions back to main
22     public static void method1() throws Exception
23     {
24         method2();
25     }
26
27     // call method3; throw exceptions back to method1
28     public static void method2() throws Exception
29     {
30         method3();
31     }
32

```

Error information  
generated by  
**getMessage** and  
**printStackTrace**



```
33     // throw Exception back to method2
34     public static void method3() throws Exception
35     {
36         throw new Exception( "Exception thrown in method3" );
37     }
38
39 } // end class Using Exceptions
```



Throw an **Exception**  
that propagates back to  
**main**

Exception thrown in method3

```
java.lang.Exception: Exception thrown in method3
    at UsingExceptions.method3(UsingExceptions.java:36)
    at UsingExceptions.method2(UsingExceptions.java:30)
    at UsingExceptions.method1(UsingExceptions.java:24)
    at UsingExceptions.main(UsingExceptions.java:11)
```



# Lab Assignment Two

---

# JAVA

---

- String Handling

- You can store alphanumeric characters within a variable using a *String* object.
- You do not use the *NULL* character to indicate the end of a Java *String* object, as you would in *C* or *C++*:
  - The *String* object itself keeps track of the number of characters it contains.
- Using the *String* class *length* method, your application can determine the number of characters a *String* object contains.

# JAVA

---

- The (+) operator lets your application append (concatenate) one string's contents to another.
- When you concatenate a numeric value, e.g. of type *int* or *float*, Java automatically calls a special function named *toString*:
  - This converts the value into a character-string.
- To simplify the output of class-member variables, you can write a *toString* function for the classes you create:
  - In the same way as for built-in types, the *toString* function for your class will be called automatically.

# JAVA

---

- String objects are immutable - that is, they cannot be changed once they've been created.
- The *java.lang* package provides a different class, *StringBuffer*, which you can use to create and manipulate character data on the fly.
- In the following example, the *reverseIt* method creates a *StringBuffer*, *dest*, the same size as *source*:
  - It then loops backwards over all the characters in *source* and appends them to *dest*, thereby reversing the string.
  - It finally converts *dest*, a *StringBuffer*, to a *String*.

# JAVA

---

```
// reverseIt() method uses two accessor methods to
// obtain information about source: charAt() & length()
class ReverseString {
    public static String reverseIt(String source) {
        int i, len = source.length();
        StringBuffer dest = new StringBuffer(len);
        for (i = (len - 1); i >= 0; i--) {
            dest.append(source.charAt(i));
        }
        return dest.toString();
    }
}
```

# JAVA

---

- Nested Classes

- » You can define a class as a member of another class. Such a class is called a nested class and is illustrated here:

```
class EnclosingClass {  
    ...  
    class ANestedClass {  
        ...  
    }  
}
```

# JAVA

---

- You use nested classes to reflect and to enforce the relationship between two classes.
  - » You should define a class within another class when the nested class makes sense only in the context of its enclosing class or when it relies on the enclosing class for its function.
  - » For example, a text cursor might make sense only in the context of a text component.
- As a member of its enclosing class, a nested class has a special privilege: It has unlimited access to its enclosing class's members, even if they are declared private.

# JAVA

---

- Like other class members, a nested class can be declared static (or not).
  - » A static nested class is called just that: a static nested class.
  - » A nonstatic nested class is called an inner class.

```
class EnclosingClass {  
    ...  
    static class StaticNestedClass {  
        ...  
    }  
    class InnerClass {  
        ...  
    }  
}
```



# JAVA

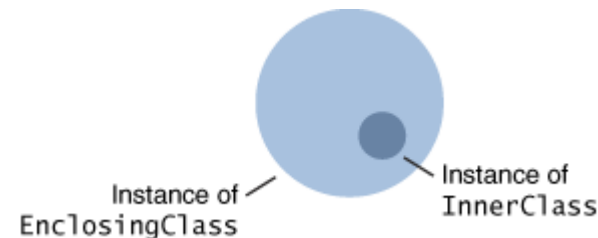
---

- As with static methods and variables, which we call class methods and variables, a static nested class is associated with its enclosing class.
  - » And like class methods, a static nested class cannot refer directly to instance variables or methods defined in its enclosing class — it can use them only through an object reference.
- As with instance methods and variables, an inner class is associated with an instance of its enclosing class and has direct access to that object's instance variables and methods.
  - » Also, because an inner class is associated with an instance, it cannot define any static members itself.

# JAVA

---

- The interesting feature about the relationship between these two classes is not that InnerClass is syntactically defined within EnclosingClass.
- Rather, it's that an instance of InnerClass can exist only within an instance of EnclosingClass and that it has direct access to the instance variables and methods of its enclosing instance.
- You may encounter nested classes of both kinds in the Java platform API and be required to use them.
- However, most nested classes that you write will probably be inner classes.



# Using Anonymous Inner Classes

- Tokenizer
  - Partition **String** into individual substrings
  - Use *delimiter*
  - Java offers `java.util.StringTokenizer`

```

1  // Fig. 10.20: TokenTest.java
2  // Testing the StringTokenizer class of the java.util package
3
4  // Java core packages
5  import java.util.*;
6  import java.awt.*;
7  import java.awt.event.*;
8
9  // Java extension packages
10 import javax.swing.*;
11
12 public class TokenTest extends JFrame {
13     private JLabel promptLabel;
14     private JTextField inputField;
15     private JTextArea outputArea;
16
17     // set up GUI and event handling
18     public TokenTest()
19     {
20         super( "Testing Class StringTokenizer" );
21
22         Container container = getContentPane();
23         container.setLayout( new FlowLayout() );
24
25         promptLabel =
26             new JLabel( "Enter a sentence and press Enter" );
27         container.add( promptLabel );
28
29         inputField = new JTextField( 20 );
30
31         inputField.addActionListener(
32
33             // anonymous inner class
34             new ActionListener() {
35

```

TokenTest.java

Line 29

inputField contains String to be  
parsed by StringTokenizer

```

36      // handle text field event
37      public void actionPerformed( ActionEvent
38      {
39          String stringToTokenize =
40              event.getActionCommand();
41          StringTokenizer tokens =
42              new StringTokenizer( stringToTokenize );
43
44          outputArea.setText( "Number of elements: " +
45              tokens.countTokens() + "\nThe tokens are:\n" );
46
47          while ( tokens.hasMoreTokens() )
48              outputArea.append( tokens.nextToken() + "\n" );
49      }
50
51      } // end anonymous inner class
52
53      ); // end call to addActionListener
54
55      container.add( inputField );
56
57      outputArea = new JTextArea( 10, 20 );
58      outputArea.setEditable( false );
59      container.add( new JScrollPane( outputArea ) );
60
61      setSize( 275, 260 ); // set the window size
62      show();              // show the window
63  }
64
65  // execute application
66  public static void main( String args[] )
67  {
68      TokenTest application = new TokenTest();
69

```

Use **StringTokenizer** to parse  
**String stringToTokenize** with  
default delimiter " \n\t\r"

Count number of tokens

Line 45

Lines 47-48

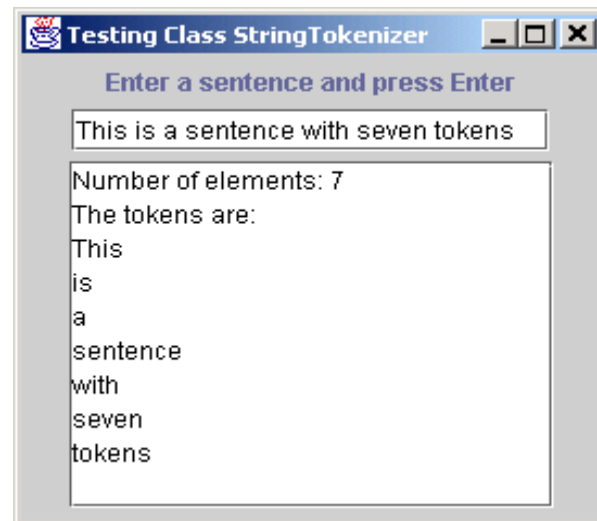
Append next token to **outputArea**, as  
long as tokens exist

```

70     application.addWindowListener(
71
72         // anonymous inner class
73         new WindowAdapter() {
74
75             // handle event when user closes window
76             public void windowClosing( WindowEvent windowEvent )
77             {
78                 System.exit( 0 );
79             }
80
81         } // end anonymous inner class
82
83     ); // end call to addWindowListener
84
85 } // end method main
86
87 } // end class TokenTest

```

TokenTest.java



# JAVA

---

- Creating and Using Packages
  - » To make types easier to find and to use, to avoid naming conflicts, and to control access, programmers bundle groups of related types into packages.
  - » Definition: A package is a collection of related types providing access protection and namespace management. Note that types refers to classes, interfaces, enums and annotations.
  - » The types that are part of the Java platform are members of various packages that bundle classes by function:
    - fundamental classes are in `java.lang`, classes for reading and writing (input and output) are in `java.io`, and so on.
  - » You can put your types in packages, too.

# JAVA

---

- Suppose that you write a group of classes that represent a collection of graphic objects, such as circles, rectangles, lines, and points. You also write an interface, Draggable, that classes implement if they can be dragged with the mouse by the user:

```
//in the Graphic.java file
public abstract class Graphic {
    ...
}
```

```
//in the Circle.java file
public class Circle extends Graphic implements Draggable {
    ...
}
```



# JAVA

---

```
//in the Rectangle.java file
public class Rectangle extends Graphic implements
    Draggable {
    . . .
}
```

```
//in the Draggable.java file
public interface Draggable {
    . . .
}
```

# JAVA

---

- You should bundle these classes and the interface in a package for several reasons:
  - » You and other programmers can easily determine that these types are related.
  - » You and other programmers know where to find types that provide graphics-related functions.
  - » The names of your types won't conflict with types names in other packages, because the package creates a new namespace.
  - » You can allow types within the package to have unrestricted access to one another yet still restrict access for types outside the package.

# JAVA

---

- Creating a Package

- » To create a package, you put a type (class, interface, enum or annotation) in it. To do this, you put a package statement at the top of the source file in which the type is defined.
- » For example, the following code appears in the source file Circle.java and puts the Circle class in the graphics package:

```
package graphics;
```

```
public class Circle extends Graphic implements Draggable {  
    ...  
}
```

# JAVA

---

- » You must include a package statement at the top of every source file that defines a class or an interface that is to be a member of the graphics package.
- » So you would also include the statement in Rectangle.java and so on:

```
package graphics;
```

```
public class Rectangle extends Graphic implements  
    Draggable {  
    . . .  
}
```

# JAVA

---

- » The scope of the package statement is the entire source file, so all classes, interfaces, enums and annotations defined in Circle.java and Rectangle.java are also members of the graphics package.
- » If you put multiple classes in a single source file, only one may be public, and it must share the name of the source file's base name.
  - Only public package members are accessible from outside the package. It's recommend that you use the convention of one public class per file, because it makes public classes easier to find and works for all compilers.
- » If you do not use a package statement, your type ends up in the default package, which is a package that has no name.

# JAVA

---

- Naming a Package

- » With programmers all over the world writing classes, interfaces, enums and annotations using the Java programming language, it is likely that two programmers will use the same name for two different classes.
- » In fact, the previous example does just that: It defines a Rectangle class when there is already a Rectangle class in the java.awt package. Yet the compiler allows both classes to have the same name. Why?
- » Because they are in different packages, and the fully qualified name of each class includes the package name.
  - That is, the fully qualified name of the Rectangle class in the graphics package is graphics.Rectangle, and the fully qualified name of the Rectangle class in the java.awt package is java.awt.Rectangle.

# JAVA

---

- » This generally works just fine unless two independent programmers use the same name for their packages.
  - What prevents this problem? Convention.
  - By Convention: Companies use their reversed Internet domain name in their package names, like this:  
`com.company.package`.
  - Name collisions that occur within a single company need to be handled by convention within that company, perhaps by including the region or the project name after the company name, for example,  
`com.company.region.package`.

# JAVA

---

- Static Import (new in J2SE 5.0)
  - » The static import feature, implemented as "import static", enables you to refer to static constants from a class without needing to inherit from it.
  - » Instead of BorderLayout.CENTER each time we add a component, we can simply refer to CENTER.

```
import static java.awt.BorderLayout.*;  
getContentPane().add(new JPanel(), CENTER);
```



# JAVA

---

- Enhanced for Loop (new in J2SE 5.0)
  - » The Iterator class is used heavily by the Collections API.
    - It provides the mechanism to navigate sequentially through a Collection.
  - » The new enhanced for loop can replace the iterator when simply traversing through a Collection as follows.
    - The compiler generates the looping code necessary and with generic types no additional casting is required.

# JAVA

---

» Before

```
ArrayList<Integer> list = new ArrayList<Integer>();  
for (Iterator i = list.iterator(); i.hasNext();)  
{  
    Integer value=(Integer)i.next();  
}
```

» After

```
ArrayList<Integer> list = new ArrayList<Integer>();  
for (Integer i : list)  
{ ... }
```

# JAVA

---

- Formatted Output (new in J2SE 5.0)
  - » Developers now have the option of using printf-type functionality to generate formatted output.
  - » Most of the common C printf formatters are available, and in addition some Java classes like Date and BigInteger also have formatting rules.
    - See the java.util.Formatter class for more information. Although the standard UNIX newline '\n' character is accepted, for cross-platform support of newlines the Java %n is recommended.

```
System.out.printf("name count%n");  
System.out.printf("%s %5d%n", user,total);
```

# JAVA

---

- Formatted Input (new in J2SE 5.0)
  - » The scanner API provides basic input functionality for reading data from the system console or any data stream.
  - » If you need to process more complex input, then there are also pattern-matching algorithms, available from the `java.util.Formatter` class.

```
Scanner s= new Scanner(System.in);  
String param= s.next();  
int value=s.nextInt();  
s.close();
```

# JAVA

---

- Varargs (new in J2SE 5.0)
  - » It requires the simple ... notation for the method that accepts the argument list and is used to implement the flexible number of arguments required for example in the printf() function.

```
void argtest(Object ... args) {  
    for (int i=0;i <args.length; i++) {  
        }  
    }
```

```
argtest("test", "data");
```

# JAVA

---

- Enumerated Types (new in J2SE 5.0)
  - » An enumerated type is a type whose legal values consist of a fixed set of constants. Common examples include compass directions, which take the values North, South, East and West and days of the week etc.
  - » In the Java programming language, you define an enumerated type by using the enum keyword. For example, you would specify a days of the week enumerated type as:

```
enum Days { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY,  
            FRIDAY, SATURDAY };
```

- » Notice that by convention the names of an enumerated type's values are spelled in uppercase letters.
- » You should use enumerated types any time you need to represent a fixed set of constants.

# JAVA

---

- Java programming language enumerated types are much more powerful than their counterparts in other languages, which are just glorified integers.
- The enum declaration defines a class (called an enum type).
- These are the most important properties of enum types:
  - » Printed values are informative.
  - » They are typesafe and they exist in their own namespace.
  - » The set of constants is not required to stay fixed for all time.
  - » You can switch on an enumeration constant.
  - » They have a static values() method that returns an array containing all of the values of the enum type in the order they are declared. This method is commonly used in combination with the for-each construct to iterate over the values of an enumerated type.
  - » You can provide methods and fields, implement interfaces, and more.

# JAVA

---

- In the following example, Planet is an enumerated type that represents the planets in the solar system.
- A Planet has constant mass and radius properties. Each enum constant is declared with values for the mass and radius parameters that are passed to the constructor when it is created.
- Note that the constructor for an enum type is implicitly private. If you attempt to create a public constructor for an enum type, the compiler displays an error message.



# JAVA

---

```
public enum Planet {
    MERCURY (3.303e+23, 2.4397e6),
    VENUS  (4.869e+24, 6.0518e6),
    EARTH  (5.976e+24, 6.37814e6),
    MARS   (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27, 7.1492e7),
    SATURN  (5.688e+26, 6.0268e7),
    URANUS  (8.686e+25, 2.5559e7),
    NEPTUNE (1.024e+26, 2.4746e7),
    PLUTO   (1.27e+22, 1.137e6);

    private final double mass; //in kilograms
    private final double radius; //in meters
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }
    public double mass() { return mass; }
    public double radius() { return radius; }

    //universal gravitational constant (m3 kg-1 s-2)
    public static final double G = 6.67300E-11;

    public double surfaceGravity() {
        return G * mass / (radius * radius);
    }
    public double surfaceWeight(double otherMass) {
        return otherMass * surfaceGravity();
    }
}
```

# JAVA

---

- In addition to its properties, Planet has methods that allow you to retrieve the surface gravity and weight of an object on each planet.
- Here is a sample program that takes your weight on earth (in any unit) and calculates and prints your weight on all of the planets (in the same unit):

```
public static void main(String[] args) {  
    double earthWeight = Double.parseDouble(args[0]);  
    double mass = earthWeight/EARTH.surfaceGravity();  
    for (Planet p : Planet.values()) {  
        System.out.printf("Your weight on %s is %f%n",  
                           p, p.surfaceWeight(mass));  
    }  
}
```

# JAVA

---

- Here's the output:

```
$ java Planet 175
Your weight on MERCURY is 66.107583
Your weight on VENUS is 158.374842
Your weight on EARTH is 175.000000
Your weight on MARS is 66.279007
Your weight on JUPITER is 442.847567
Your weight on SATURN is 186.552719
Your weight on URANUS is 158.397260
Your weight on NEPTUNE is 199.207413
Your weight on PLUTO is 11.703031
```

- There's one limitation of enum types: although enum types are classes, you cannot define a hierarchy of enums. In other words, it's not possible for one enum type to extend another enum type.

# Lab Assignment Three

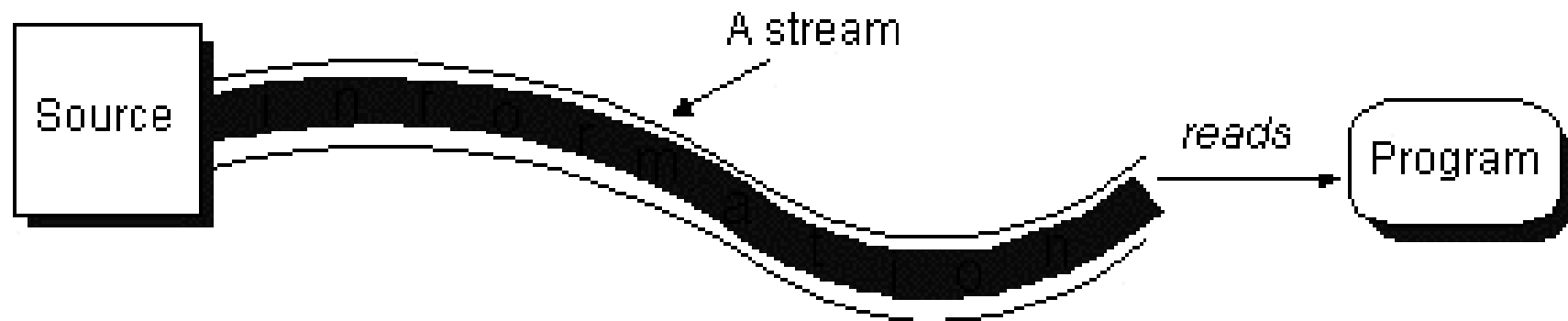
---

# IO Streams

---

- IO Streams

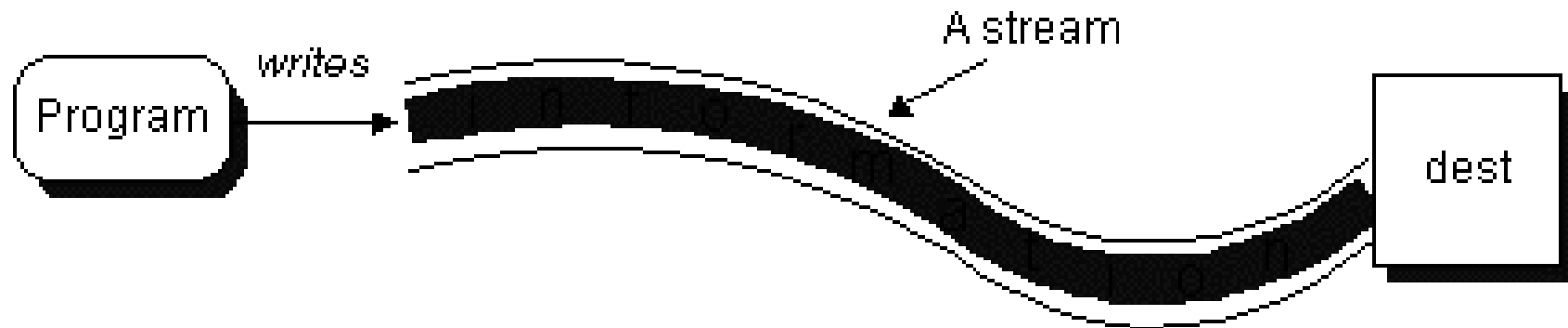
- To bring in information, a program opens a stream on an information source (a file, memory, a socket) and reads the information serially, like this:



# IO Streams

---

- Similarly, a program can send information to an external destination by opening a stream to a destination and writing the information out serially, like this:



# IO Streams

---

- No matter where the information is coming from or going to and no matter what type of data is being read or written, the algorithms for reading and writing data are usually the same.

## Reading

open a stream  
while more information  
    read information  
close the stream

## Writing

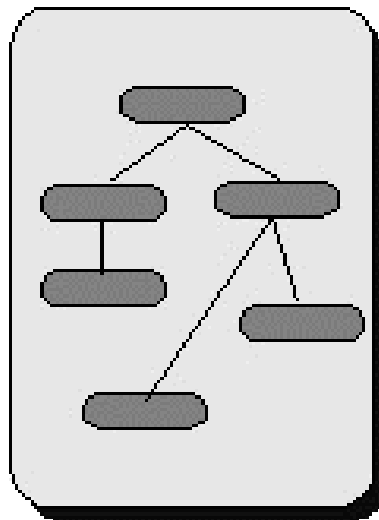
open a stream  
while more information  
    write information  
close the stream

# IO Streams

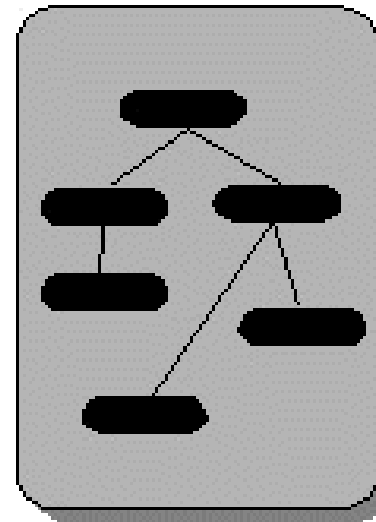
---

- The java.io package contains a collection of stream classes that support reading and writing streams.
- Divided into two class hierarchies based on the data (either characters or bytes) on which they operate.

Character Streams



Byte Streams

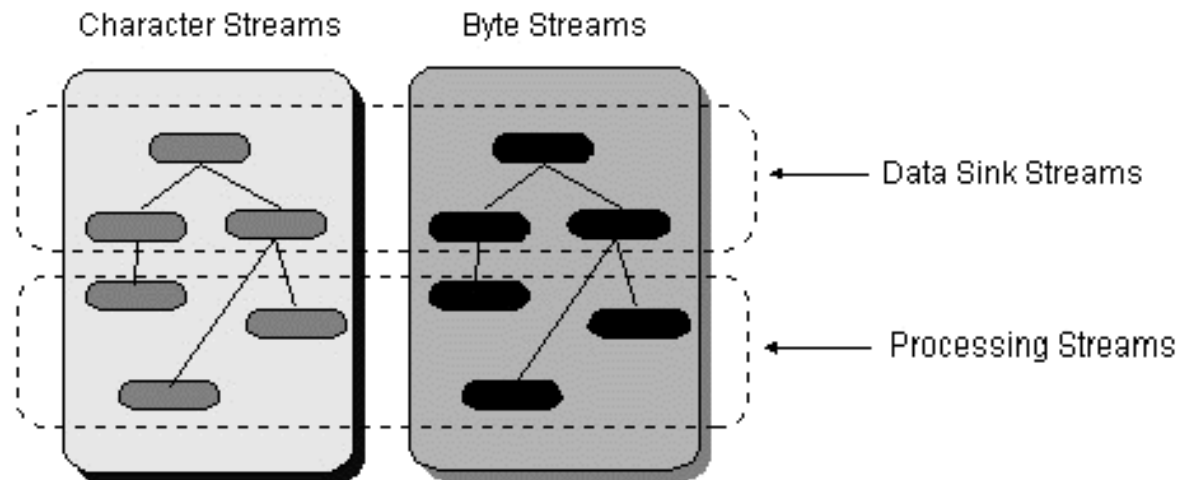




# IO Streams

---

- However, it's often more convenient to group the classes based on their purpose rather than on the data type they read and write.
- Thus, we can cross-group the streams by whether they read from and write to data "sinks" or process the information as its being read or written.



# IO Streams

---

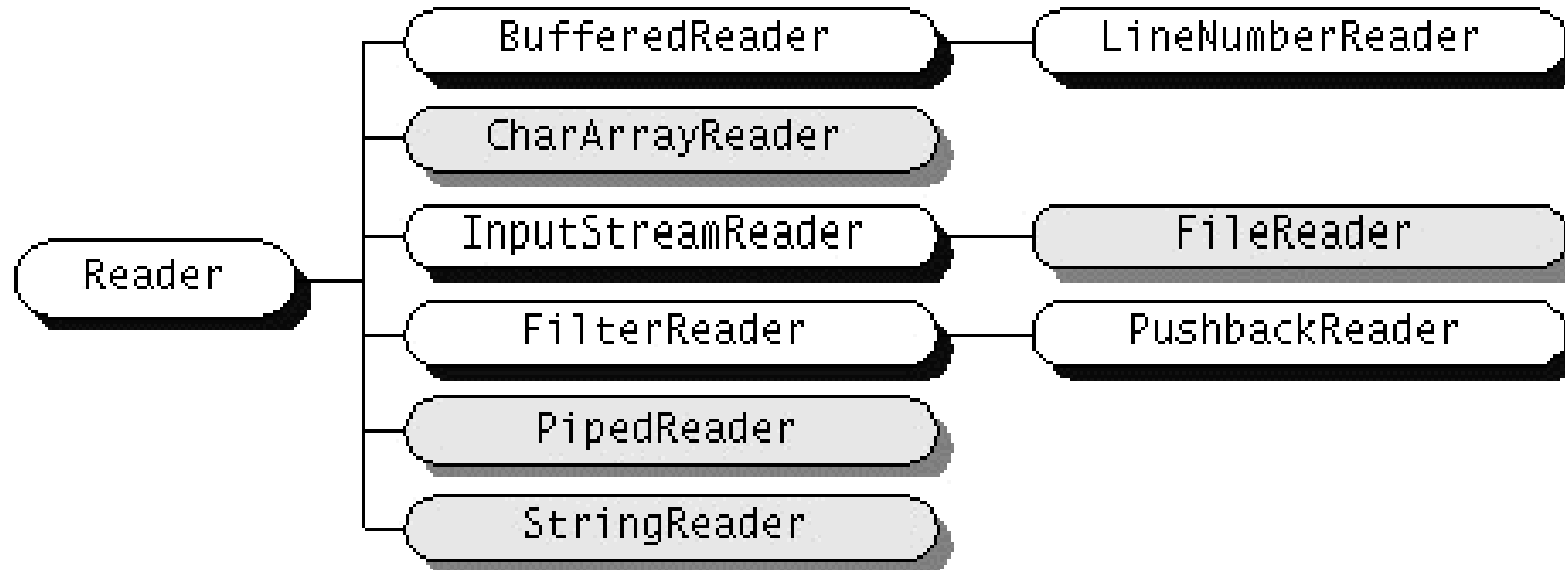
## » Character Streams

- Reader and Writer are the abstract super-classes for character streams in java.io.\*
- Reader provides the API and partial implementation for readers - streams that read 16-bit unicode characters.
- Writer provides the API and partial implementation for writers - streams that write 16-bit characters.
- Subclasses of Reader and Writer implement specialised streams.

# IO Streams

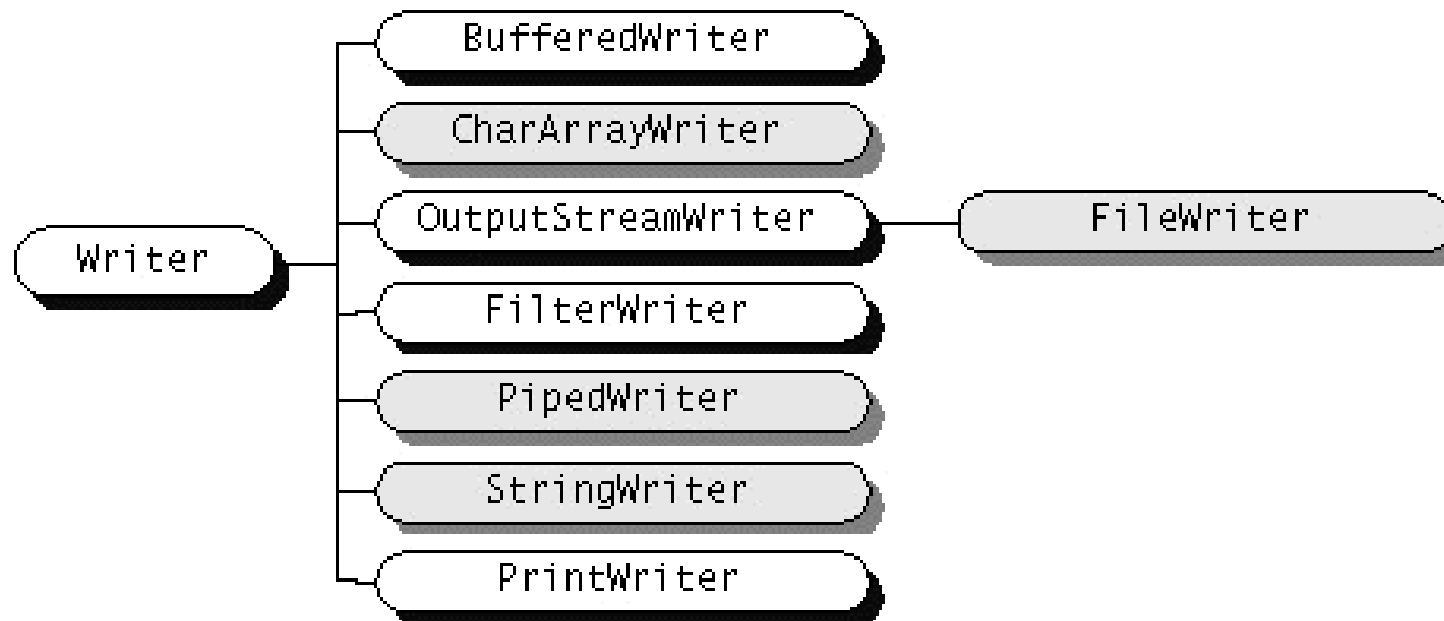
---

- Class hierarchies for the Reader and Writer classes
  - Those that read from or write to data sinks are shown in grey in the following figures, those that perform some sort of processing are shown in white.



# IO Streams

---



- Most programs should use readers and writers to read and write information.
  - This is because they both can handle any character in the Unicode character set (while the byte streams are limited to ISO-Latin-1 8-bit bytes).

# IO Streams

---

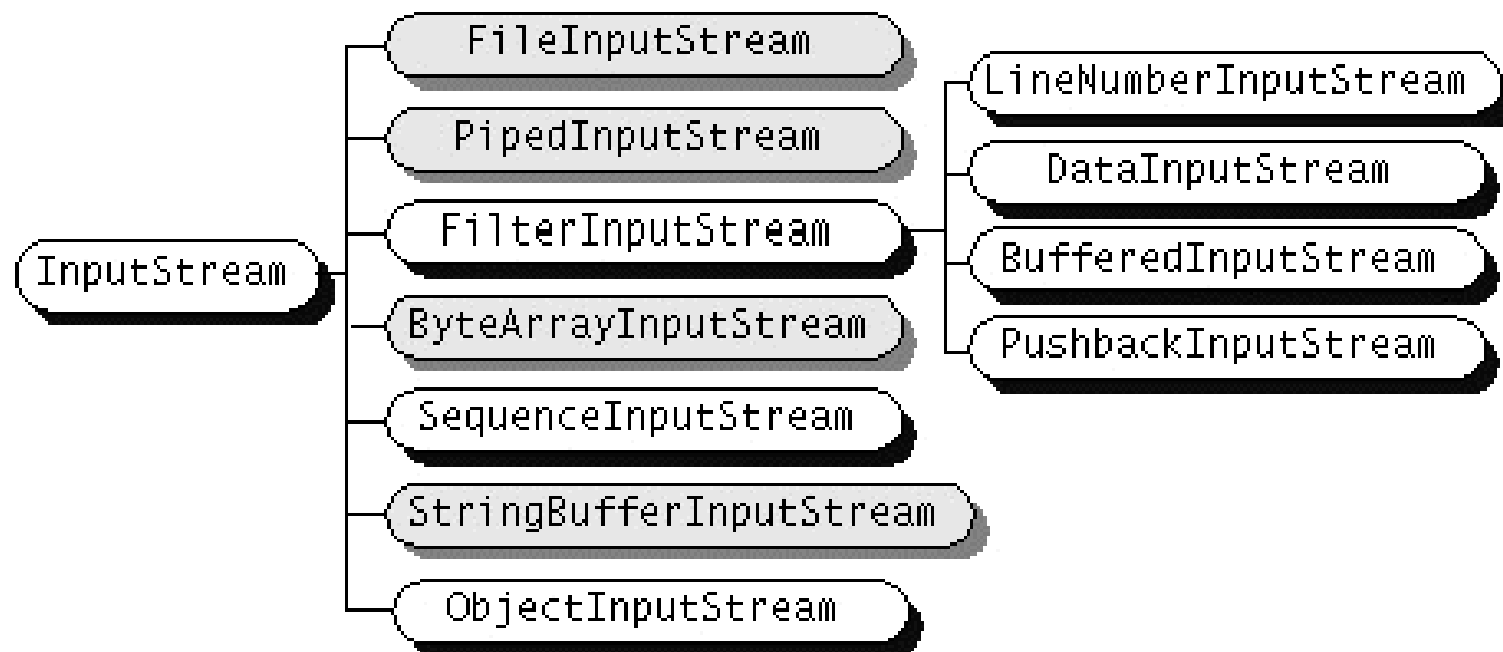
## » Byte Streams

- Programs should use the byte streams, descendants of `InputStream` and `OutputStream`, to read and write 8-bit bytes.
- `InputStream` and `OutputStream` provide the API and some implementation for input streams (streams that read 8-bit bytes) and output streams (streams that write 8-bit bytes).
- These streams are typically used to read and write binary data such as images and sounds.

# IO Streams

---

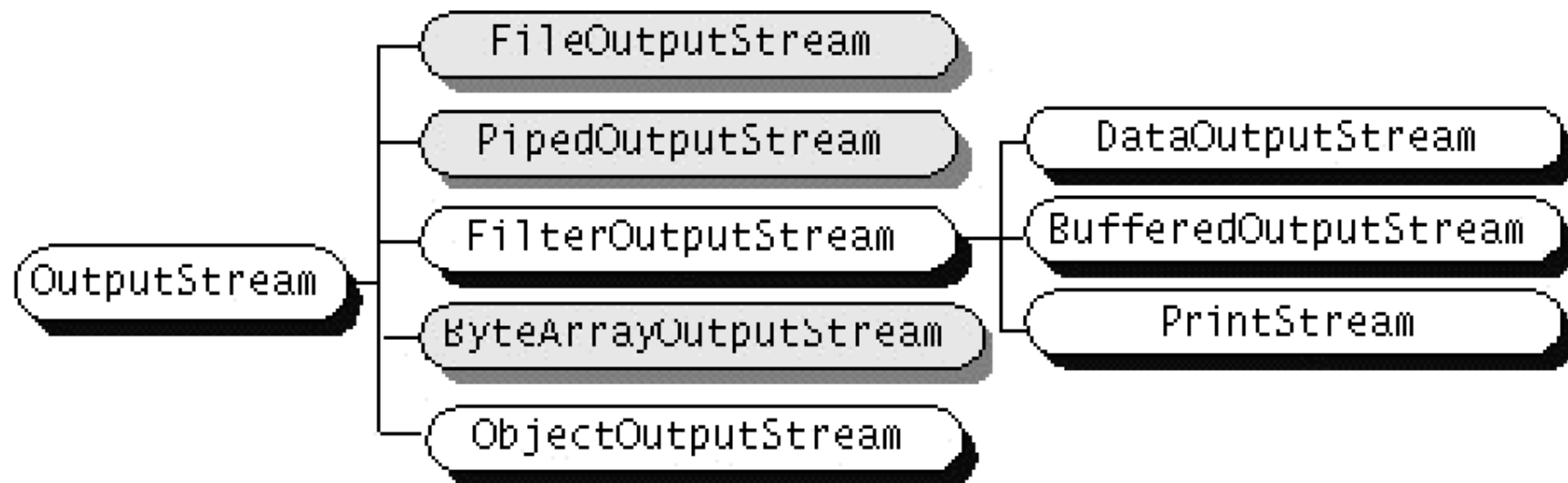
- As with Reader and Writer, subclasses of InputStream and OutputStream provide specialised I/O:



# IO Streams

---

- Note that these also fall into two categories:
  - Data sink streams and processing streams (again shown in grey and white):



# IO Streams

---

## » Data Sink Streams

- Data sink streams read from or write to specialised data sinks such as strings, files, or pipes.
- Typically, for each reader or input stream intended to read from a specific kind of input source, java.io contains a parallel writer or output stream that can create it.
- Note that both the character stream group and the byte stream group contain parallel pairs of classes that operate on the same data sinks.



# IO Streams

---

<u>Sink Type</u>	<u>Character Streams</u>	<u>Byte Streams</u>
Memory	CharArrayReader CharArrayWriter StringReader StringWriter	ByteArrayInputStream ByteArrayOutputStream StringBufferInputStream
Pipe	PipedReader PipedWriter	PipedInputStream PipedOutputStream
File	FileReader FileWriter	FileInputStream FileOutputStream

# IO Streams

---

## » Processing Streams

- Processing streams perform some sort of operation, such as buffering or data conversion, as they read and write.
- Like the data sink streams, `java.io` often contains pairs of streams:
  - One that performs a particular operation during reading and another that performs the same operation (or reverses it) during writing.
  - Note also that in many cases, `java.io` contains character streams and byte streams that perform the same processing but for the different data type.

# IO Streams

---

<u>Process</u>	<u>Character Streams</u>	<u>Byte Streams</u>
Buffering	BufferedReader BufferedWriter	BufferedInputStream BufferedOutputStream
Filtering	FilterReader FilterWriter	FilterInputStream FilterOutputStream
Stream Conversion	InputStreamReader OutputStreamWriter	
Concatenation		SequenceInputStream

# IO Streams

---

<u>Process</u>	<u>Character Streams</u>	<u>Byte Streams</u>
Data Conversion		DataInputStream, DataOutputStream
Counting	LineNumberReader	LineNumberInputStream
Peeking Ahead	PushbackReader	PushbackInputStream
Printing	PrintWriter	PrintStream
Object Serialization	ObjectInputStream	ObjectOutputStream

# Lab Assignment Four

---

# Object Serialization

---

- Object Serialization

- Two streams in java.io - `ObjectInputStream` and `ObjectOutputStream` - are special in that they can read and write actual objects.
- The key to writing an object is to represent its state in a serialized form sufficient to reconstruct the object as it is read.
- Thus reading and writing objects is a process called object serialization.
- Object serialization can be useful in lots of application domains.

# Object Serialization

---

- You can use object serialization in the following ways:
  - Remote Method Invocation (RMI) - communication between objects via sockets i.e. to pass various objects back and forth between the client and server.
  - Lightweight persistence - the archival of an object for use in a later invocation of the same program.
- As a Java programmer, you need to know about object serialization from two points of view:
  - How to serialize existing objects.
  - How to provide serialization for new classes.

# Object Serialization

---

## » Serializing Objects

- Reconstructing an object from a stream requires that the object first be written to a stream.
- Writing objects to a stream is a straight-forward process.
- For example, the following code sample gets the current time in milliseconds by constructing a Date object and then serializes that object:
  - ObjectOutputStream is a processing stream, so it must be constructed on another stream.



# Object Serialization

---

```
FileOutputStream out = new FileOutputStream("theTime");  
ObjectOutputStream s = new ObjectOutputStream(out);  
s.writeObject("Today");  
s.writeObject(new Date());  
s.flush();
```

- This code constructs an ObjectOutputStream on a FileOutputStream, thereby serializing the object to a file named theTime.
- Next, the string Today and a Date object are written to the stream with the writeObject method of ObjectOutputStream.

# Object Serialization

---

- If an object refers to other objects, then all of the objects that are reachable from the first must be written at the same time so as to maintain the relationships between them.
- Thus the `writeObject` method serializes the specified object, traverses its references to other objects recursively, and writes them all.
  - The `writeObject` method throws a `NotSerializableException` if it's given an object that is not serializable.
  - An object is serializable only if its class implements the `Serializable` interface.

# Object Serialization

---

- » Reading from an `ObjectInputStream`
  - Once you've written objects and primitive data types to a stream, you'll likely want to read them out again and reconstruct the objects.
  - Here's code that reads in the `String` and the `Date` object that was written to the file named `theTime` in the last example:

```
FileInputStream in = new FileInputStream("theTime");
ObjectInputStream s = new ObjectInputStream(in);
String today = (String)s.readObject();
Date date = (Date)s.readObject();
```

# Object Serialization

---

- Like `ObjectOutputStream`, `ObjectInputStream` must be constructed on another stream.
- In this example, the objects were archived in a file, so the code constructs an `ObjectInputStream` on a `FileInputStream`.
- Next, the code uses `ObjectInputStream`'s `readObject` method to read the `String` and the `Date` objects from the file.
- The objects must be read from the stream in the same order in which they were written.

# Object Serialization

---

- Note that the return value from `readObject` is an object that is cast to and assigned to a specific type.
- The `readObject` method deserializes the next object in the stream and traverses its references to other objects recursively to deserialize all objects that are reachable from it.
- In this way, it maintains the relationships between the objects.
- The methods in `DataInput` parallel those defined in `DataOutput` for writing primitive data types.

# Object Serialization

---

## » Providing Serialization for Your Own Classes

- An object is serializable only if its class implements the Serializable interface.
- Thus, if you want to serialize the instances of one of your classes, the class must implement the Serializable interface.
- The good news is that Serializable is an empty interface.
- That is, it doesn't contain any method declarations; its purpose is simply to identify classes whose objects are serializable.

# Object Serialization

---

- Here's the complete definition of the Serializable interface:

```
package java.io;  
public interface Serializable {  
    // there's nothing in here!  
};
```

- To make instances of your classes serializable, just add the implements Serializable clause to your class declaration.

# Object Serialization

---

- Example of serializable class ...

```
public class MySerializableClass implements  
    Serializable {  
    ...  
}
```

- You don't have to write any methods.
- The serialization of instances of this class are handled by the defaultWriteObject method of ObjectOutputStream.



# Object Serialization

---

- This method automatically writes out everything required to reconstruct an instance of the class, including the following:
  - Class of the object
  - Class signature
  - Values of all non-transient and non-static members, including members that refer to other objects.
- For many classes, the default behaviour is fine.
- However, default serialization can be slow, and a class might want more explicit control over the serialization.

# Object Serialization

---

## » Customising Serialization

- You can customise serialization for your classes by providing two methods for it: *writeObject* and *readObject*.
- The *writeObject* method controls what information is saved and is typically used to append additional information to the stream.
- The *readObject* method either reads the information written by the corresponding *writeObject* method or can be used to update the state of the object after it has been restored.

# Object Serialization

---

- The writeObject and readObject methods must be declared exactly as shown in the following example.
- Also, it should call the stream's defaultWriteObject as the first thing it does to perform default serialization (any special arrangements can be handled afterwards):

```
private void writeObject(ObjectOutputStream s)
    throws IOException {
    s.defaultWriteObject();
    // customised serialization code
}
```

# Object Serialization

---

- The readObject method must read in everything written by writeObject in the same order in which it was written.
- Here's the readObject method that corresponds to the writeObject method just shown:

```
private void readObject(ObjectInputStream s)
    throws IOException {
    s.defaultReadObject();
    // customised deserialization code
    // followed by code to update the object, if necessary
}
```

# Object Serialization

---

- Also, the `readObject` method can perform calculations or update the state of the object in some way.
- The `writeObject` and `readObject` methods are responsible for serializing only the immediate class.
- Any serialization required by the superclasses is handled automatically.
  - However, a class that needs to explicitly co-ordinate with its superclasses to serialize itself can do so by implementing the `Externalizable` interface.

# Object Serialization

---

## » Externalizable Interface

- For complete, explicit control of the serialization process, a class must implement the Externalizable interface.
- For Externalizable objects, only the identity of the object's class is automatically saved by the stream.
- The class is responsible for writing and reading its contents, and it must co-ordinate with its superclasses to do so.

# Random File Access

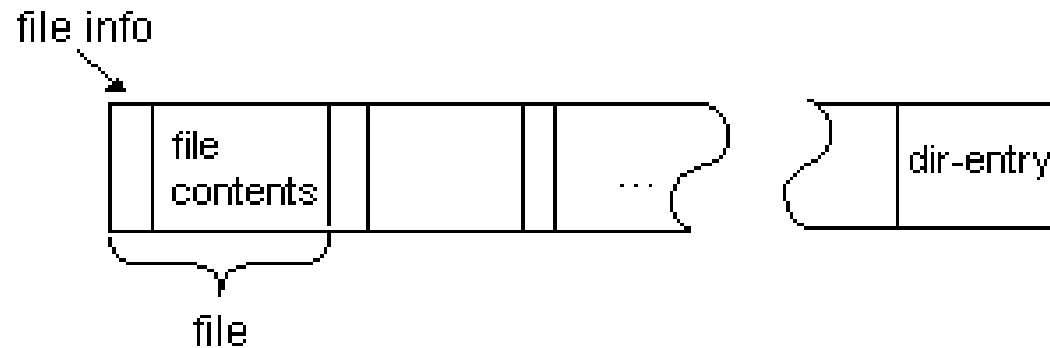
---

- Working with Random Access Files
  - The input and output streams in this lesson so far have been sequential access streams:
    - Streams whose contents must be read or written sequentially.
  - While still incredibly useful, sequential access files are a consequence of a sequential medium such as magnetic tape.
  - Random access files, on the other hand, permit non-sequential, or random, access to the contents of a file.

# Random File Access

---

- Consider the archive format known as "zip."
  - Zip archives contain files and are typically compressed to save space.
- Zip archives also contain a dir-entry at the end that indicates where the various files contained within the zip archive begin:





# Random File Access

---

- Suppose that you want to extract a specific file from a zip archive.
- If you use a sequential access stream, you have to do the following:
  - Open the zip archive.
  - Search through the zip archive until you located the file you wanted to extract.
  - Extract the file.
  - Close the zip archive.
- On average, using this algorithm, you'd have to read half the zip archive before finding the file that you wanted to extract.

# Random File Access

---

- You can extract the same file from the zip archive more efficiently using the seek feature of a random access file:
  - Open the zip archive.
  - Seek to the dir-entry and locate the entry for the file you want to extract from the zip archive.
  - Seek (backwards) within the zip archive to the position of the file to extract.
  - Extract the file and close the zip archive.
- This algorithm is more efficient because you only read the dir-entry and the file that you want to extract.

# Random File Access

---

- The `RandomAccessFile` class in the `java.io` package implements a random access file.
- Unlike the input and output stream classes in `java.io`, `RandomAccessFile` is used for both reading and writing files.
  - The `RandomAccessFile` class implements both the `DataInput` and `DataOutput` interfaces and therefore can be used for both reading and writing.
- You create a `RandomAccessFile` object with different arguments depending on whether you intend to read or write.

# Random File Access

---

- RandomAccessFile is similar to FileInputStream and FileOutputStream in that you specify a file on the native file system to open.
  - You can do this with a filename or a File object.
- When you create a RandomAccessFile, you must indicate whether you will be just reading the file or also writing to it.
- The code creates a RandomAccessFile to read the file named farrago.txt:  
`new RandomAccessFile("farrago.txt", "r");`

# Random File Access

---

- This code opens the same file for both reading and writing:  

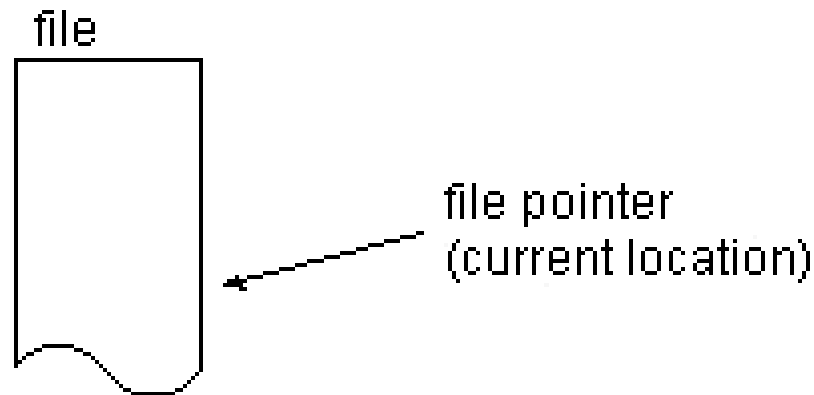
```
new RandomAccessFile("farrago.txt", "rw");
```
- After the file has been opened, you can use the common readXXX or writeXXX methods to perform I/O on the file.
- The RandomAccessFile class supports the notion of a file pointer.
  - This pointer indicates the current location in the file.
- When the file is first created, the file pointer is 0, indicating the beginning of the file.

# Random File Access

---

- Calls to the readXXX and writeXXX methods adjust the file pointer by the number of bytes read or written e.g.

```
int data = file.readInt();  
file.writeInt(data);
```



# Random File Access

---

- In addition to the normal file I/O methods that implicitly move the file pointer when the operation occurs, `RandomAccessFile` also contains three methods for explicitly manipulating the file pointer.
  - `skipBytes()` - moves the file pointer forward the specified number of bytes.
  - `seek()` - positions the file pointer just before the specified byte.
  - `getFilePointer()` - returns the current byte location of the file pointer.

# Random File Access

---

- `RandomAccessFile` is somewhat disconnected from the input and output streams in `java.io` - it doesn't inherit from the `InputStream` or `OutputStream`.
- This has some disadvantages in that you can't apply the same filters to `RandomAccessFiles` that you can to streams.
- However, `RandomAccessFile` does implement the `DataInput` and `DataOutput` interfaces:
  - If you design a filter that works for either `DataInput` or `DataOutput`, it will work on any `RandomAccessFile`.



# Lab Assignment Five

---

# Collections Framework

---

- What Is a Collection?

- A collection (sometimes called a container) is simply an object that groups multiple elements into a single unit.
- Collections are used to store, retrieve and manipulate data, and to transmit data from one method to another.
- Collections typically represent data items that form a natural group:
  - Like a poker hand (a collection of cards), a mail folder (a collection of letters), or a telephone directory (a collection of name-to-phone-number mappings).

# Collections Framework

---

- If you've used Java -- or just about any other programming language -- you're already familiar with collections.
- Collection implementations in earlier versions of Java included Vector , Hashtable , and array.
  - Vector and Hashtable are part of the java.util package and are widely used in existing code.
- While earlier versions of Java contained collection implementations, they did not contain a full collections framework.

# Collections Framework

---

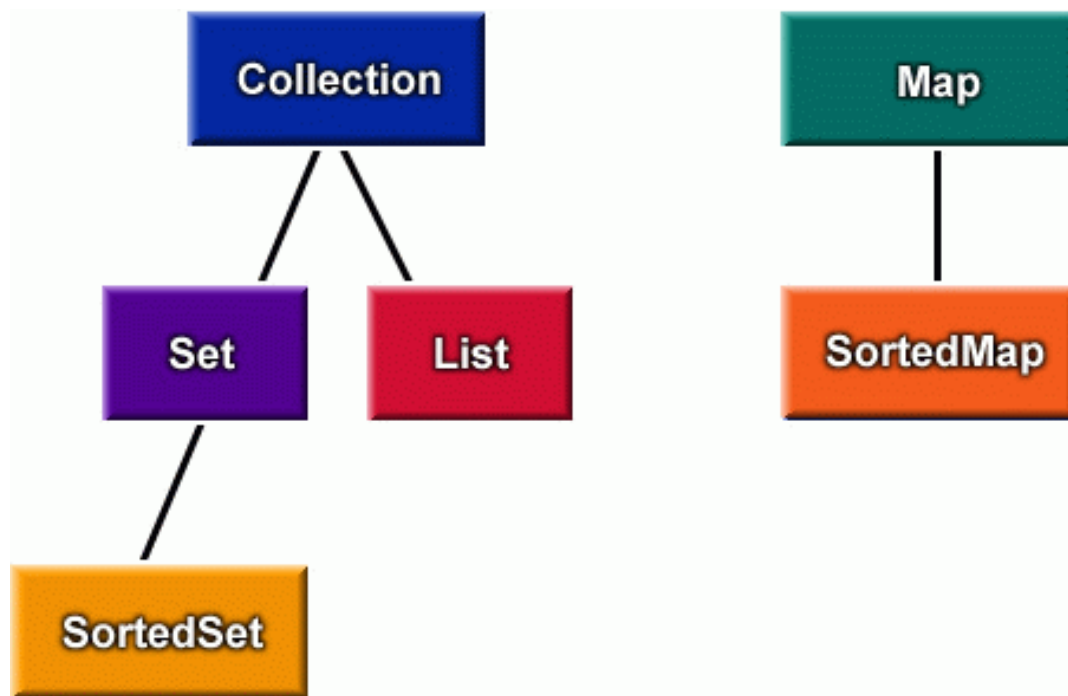
- Interfaces

- The core collection interfaces are the interfaces used to manipulate collections, and to pass them from one method to another.
- The basic purpose of these interfaces is to allow collections to be manipulated independently of the details of their representation.
- The core collection interfaces are the main foundation of the collections framework.

# Collections Framework

---

– Core Collections Interfaces



# Collections Framework

---

- The core collection interfaces form a hierarchy:
  - A Set is a special kind of Collection, and a SortedSet is a special kind of Set, and so forth.
  - Note also that the hierarchy consists of two distinct trees: a Map is not a true Collection.
- To keep the number of core collection interfaces manageable, the JDK doesn't provide separate interfaces for each variant of each collection type.
- Among the possible variants are immutable, fixed-size, and append-only.

# Collections Framework

---

- Instead, the modification operations in each interface are designated optional:
  - A given implementation may not support some of these operations.
  - If an unsupported operation is invoked, a collection throws an `UnsupportedOperationException` .
- Implementations are responsible for documenting which of the optional operations they support.
  - All of the JDK's general purpose implementations support all of the optional operations.

# Collections Framework

---

## » Collection Interface

- The Collection interface is the root of the collection hierarchy.
- A Collection represents a group of objects, known as its elements.
- Some Collection implementations allow duplicate elements and others do not.
- Some are ordered and others unordered.
  - The JDK doesn't provide any direct implementations of this interface: It provides implementations of more specific sub-interfaces like Set and List.



# Collections Framework

---

- This interface is the lowest common denominator that all collections implement.
- Collection is used to pass collections around and manipulate them when maximum generality is desired.

```
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();
```

# Collections Framework

---

// 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[]);

}

# Collections Framework

---

## » Iterator Interface

- The object returned by the `Collection.iterator()` method deserves special mention.
- It is an `Iterator`, which is very similar to an `Enumeration`, but differs in two respects:
  - `Iterator` allows the caller to remove elements from the underlying collection during the iteration with well-defined semantics.
  - Method names have been improved.
- The first point is important: There was no safe way to remove elements from a collection while traversing it with an `Enumeration`.

# Collections Framework

---

- The semantics of this operation were ill-defined, and differed from implementation to implementation.
- The Iterator interface is shown below:

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

# Collections Framework

---

- The hasNext method is identical in function to Enumeration.hasMoreElements, and the next method is identical in function to Enumeration.nextElement.
- The remove method removes from the underlying Collection the last element that was returned by next.
  - The remove method may be called only once per call to next, and throws an exception if this is violated.
  - Note that Iterator.remove is the only safe way to modify a collection during iteration; the behaviour is unspecified if the underlying collection is modified in any other way while the iteration is in progress.

# Collections Framework

---

- The following sample shows you how to use an Iterator to filter a Collection, that is, to traverse the collection, removing every element that does not satisfy some condition:

```
static void filter(Collection c) {  
    for (Iterator i = c.iterator(); i.hasNext(); )  
        if (!cond(i.next()))  
            i.remove();  
}  
// cond() is some other related method which checks  
// the element for some condition ...
```

# Collections Framework

---

- Two things should be kept in mind when looking at this simple piece of code:
  - The code is polymorphic: it works for any Collection that supports element removal, regardless of implementation.
  - This example shows how easy it is to write a polymorphic algorithm under the collections framework!
  - It would have been impossible to write this using Enumeration instead of Iterator, because there's no safe way to remove an element from a collection while traversing it with an Enumeration.

# Collections Framework

---

## » Set Interface

- A Set is a collection that cannot contain duplicate elements.
- As you might expect, this interface models the mathematical set abstraction.
- It is used to represent sets like the cards comprising a poker hand, the courses making up a student's schedule, or the processes running on a machine.
- The Set interface has the same methods as the Collection interface.



# Collections Framework

---

- The JDK contains two general-purpose Set implementations.
  - HashSet, which stores its elements in a hash table, is the best-performing implementation.
  - TreeSet, which stores its elements in a tree format, and guarantees the order of iteration.
- There follows a program that takes the words in its argument list and prints out:
  - Any duplicate words.
  - The number of distinct words.
  - A list of the words with duplicates eliminated.

# Collections Framework

---

```
import java.util.*;

public class FindDups {
    public static void main(String args[]) {
        Set s = new HashSet();
        for (int i=0; i<args.length; i++)
            if (!s.add(args[i]))
                System.out.println("Duplicate detected: "+args[i]);

        System.out.println(s.size()+" distinct words detected: "+s);
    }
}
```

# Collections Framework

---

- Running the program produces the following:

```
% java FindDups i came i saw i left
```

```
Duplicate detected: i
```

```
Duplicate detected: i
```

```
4 distinct words detected: [came, left, saw, i]
```

- Note that the example code always refers to the collection by its interface type (Set), rather than by its implementation type (HashSet).

# Collections Framework

---

- Using an interface data type (like Set) is a strongly recommended programming practice:
  - It gives you the flexibility to change implementations merely by changing the constructor.
- If the variables used to store a collection, or the parameters used to pass it around, are declared to be of the collection's implementation type rather than its interface type:
  - Then all such variables and parameters must be changed to change the collection's implementation type.

# Collections Framework

---

## » List Interface

- A List is an ordered collection (sometimes called a sequence).
- Lists can contain duplicate elements.
- The user of a List generally has precise control over where in the List each element is inserted.
- The user can access elements by their integer index (position).
- If you've used Vector, you're already familiar with the general flavour of List.

# Collections Framework

---

– The List interface is shown below:

```
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
```

// Continued on next slide ...

# Collections Framework

---

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

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

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

# Collections Framework

---

- The JDK contains two general-purpose List implementations.
  - ArrayList, which is generally the best-performing implementation.
  - LinkedList which offers better performance under certain circumstances.
- Also, the standard `java.util.Vector` class has been retrofitted to implement List in new versions of the JDK (1.2 and higher).
- The operations inherited from Collection all do what you'd expect them to do.



# Collections Framework

---

## » Map Interface

- A Map is an object that maps keys to values.
- Maps cannot contain duplicate keys:
  - Each key can map to at most one value.
  - If you've used Hashtable, you're already familiar with the general flavour of Map.
- The last two core collection interfaces (SortedSet and SortedMap) are merely sorted versions of Set and Map.
  - In order to understand these interfaces, you have to know how order is maintained among objects.

# Collections Framework

---

## » Object Ordering

- There are two ways to order objects:
  - The Comparable interface provides automatic natural order on classes that implement it.
  - While the Comparator interface gives the programmer complete control over object ordering.
- Note that these are not core collection interfaces, but underlying infrastructure.
  - E.g. the Comparable interface is implemented by classes like Byte, Float, Integer etc
- This interface imposes a total ordering on the objects of each class that implements it.

# Collections Framework

---

## » Sorted Set and Map Interfaces

- A SortedSet is a Set that maintains its elements in ascending order.
  - Several additional operations are provided to take advantage of the ordering.
  - The SortedSet interface is used for things like word lists and membership rolls.
- A SortedMap is a Map that maintains its mappings in ascending key order.
  - It is the Map analogue of SortedSet.
  - The SortedMap interface is used for apps like dictionaries and telephone directories.

# Collections Framework

---

- Collection Implementation Classes
  - The general-purpose JDK implementation classes are summarised in the table below:

<u>Interface</u>	<u>Class</u>	<u>Underlying Data Structure</u>
Set	HashSet	Hash Table
	TreeSet	Balanced Tree
List	ArrayList	Resizable Array
	LinkedList	Linked List
Map	HashMap	Hash Table
	TreeMap	BalancedTree

# Collections Framework

---

- All of the classes implement all the optional operations contained in their interfaces.
- All permit null elements, keys and values.
- Each one is unsynchronised:
  - If you need a synchronised collection, so called *synchronisation wrappers*, allow any collection to be transformed into a synchronised collection.
- All have fail-fast iterators, which detect illegal concurrent modification during iteration and fail quickly and cleanly.
  - Rather than risking arbitrary, non-deterministic behaviour at an undetermined time in the future.

# Lab Assignment Six

---

# Collections Framework

---

- Algorithms

- Polymorphic algorithms are pieces of reusable functionality provided by the JDK.
  - All of them come from the `java.util.Collections` class.
- All take the form of static methods whose first argument is the collection on which the operation is to be performed.
- The great majority of the algorithms provided by the Java platform operate on List objects, but a couple of them (min and max) operate on arbitrary Collection objects.

# Collections Framework

---

## » Sorting

- The **sort** algorithm reorders a List so that its elements are ascending order according to some ordering relation.
- Two forms of the operation are provided.
- The simple form just takes a List and sorts it according to its elements' natural ordering.
- The sort operation uses a slightly optimised merge sort algorithm.



# Collections Framework

---

- The important things to know about the merge sort algorithm are that it is:
- *Fast*: This algorithm is guaranteed to run in  $n \log(n)$  time, and runs substantially faster on nearly sorted lists.
  - Empirical studies showed it to be as fast as a highly optimised quicksort.
  - Quicksort is generally regarded to be faster than merge sort, but isn't stable, and doesn't guarantee  $n \log(n)$  performance.

# Collections Framework

---

- Stable: That is to say, it doesn't reorder equal elements.
  - This is important if you sort the same list repeatedly on different attributes.
  - If a user of a mail program sorts his in-box by mailing date, and then sorts it by sender, the user naturally expects that the now-contiguous list of messages from a given sender will (still) be sorted by mailing date.
  - This is only guaranteed if the second sort was stable.

# Collections Framework

---

- Here's a small program that prints out its arguments in lexicographic (alphabetical) order.

```
import java.util.*;

public class Sort {
    public static void main(String args[]) {
        List l = Arrays.asList(args);
        Collections.sort(l);
        System.out.println(l);
    }
}
```

# Collections Framework

---

- Running the program produces the following:

```
% java Sort i walk the line
```

```
[i, line, the, walk]
```

- The second form of sort takes a Comparator in addition to a List and sorts the elements with the Comparator.
  - Comparators can be passed to a sort method (such as Collections.sort) to allow precise control over the sort order.

# Collections Framework

---

## » Shuffling

- The ***shuffle*** algorithm does the opposite of what sort does: it destroys any trace of order that may have been present in a List.
  - That is to say, it reorders the List, based on input from a source of randomness, such that all possible permutations occur with equal likelihood (assuming a fair source of randomness).
- This algorithm is useful in implementing games of chance.
  - For example, it could be used to shuffle a List of Card objects representing a deck.

# Collections Framework

---

- Shuffle can also be useful for generating test cases.
- There are two forms of this operation.
- The first just takes a List and uses a default source of randomness.
- The second requires the caller to provide a Random object to use as a source of randomness.

# Collections Framework

---

## » Routine Data Manipulation

- The Collections class provides three algorithms for doing routine data manipulation on List objects.
- All of these algorithms are pretty straightforward:
- ***reverse***: Reverses the order of the elements in a List.
- ***fill***: Overwrites every element in a List with the specified value.
  - This operation is useful for re-initialising a List.

# Collections Framework

---

- ***copy***: Takes two arguments, a destination List and a source List, and copies the elements of the source into the destination, overwriting its contents.
  - The destination List must be at least as long as the source.
  - If it is longer, the remaining elements in the destination List are unaffected.



# Collections Framework

---

## » Searching

- The ***binarySearch*** algorithm searches for a specified element in a sorted List using the binary search algorithm.
- There are two forms of this algorithm.
- The first takes a List and an element to search for (the "search key").
- This form assumes that the List is sorted into ascending order according to the natural ordering of its elements.

# Collections Framework

---

- The second form of the call takes a Comparator in addition to the List and the search key, and assumes that the List is sorted into ascending order according to the specified Comparator.
- The sort algorithm (described already) can be used to sort the List prior to calling `binarySearch`.
- The return value is the same for both forms:
  - If the List contains the search key, its index is returned.
  - Otherwise, the return value is  $-(\text{insertion point}) - 1$ .

# Collections Framework

---

- In the negative case, the insertion point is defined as the point at which the value would be inserted into the List:
  - The index of the first element greater than the value, or `list.size()` if all elements in the List are less than the specified value.
  - This admittedly ugly formula was chosen to guarantee that the return value will be  $\geq 0$  if and only if the search key is found.
  - It's basically a hack to combine a boolean ("found") and an integer ("index") into a single int return value.

# Collections Framework

---

- The following idiom, usable with both forms of the `binarySearch` operation, looks for the specified search key, and inserts it at the appropriate position if it's not already present:

```
int pos = Collections.binarySearch(l, key);  
if (pos < 0)  
    l.add(-pos-1, key);
```

# Collections Framework

---

## » Finding Extreme Values

- The ***min*** and ***max*** algorithms return, respectively, the minimum and maximum element contained in a specified Collection.
- Both of these operations come in two forms.
  - The simple form takes only a Collection, and returns the minimum (or maximum) element according to the elements' natural ordering.
  - The second form takes a Comparator in addition to the Collection and returns the minimum (or maximum) element according to the specified Comparator.

# Collections Framework

---

- These are the only algorithms provided by the Java platform that work on arbitrary Collection objects, as opposed to List objects.
- Like the fill algorithm, described earlier, these algorithms are quite straightforward to implement.
- They are included in the Java platform solely as a convenience to programmers.
- Most programmers will probably never need to implement their own collections classes, but this can be done if necessary.

# Collections Framework

---

- Generic Types (new in J2SE 5.0)
  - » Generic types have been widely anticipated by the Java Community and are now part of J2SE 5.0.
  - » One of the first places to see generic types in action is the Collections API.
  - » The Collections API provides common functionality like LinkedLists, ArrayLists and HashMaps that can be used by more than one Java type.
  - » The next example uses the 1.4.2 libraries and the default javac compile mode:

```
ArrayList list = new ArrayList();  
list.add(0, new Integer(42));  
int total = ((Integer)list.get(0)).intValue();
```

# Collections Framework

---

- » The cast to Integer on the last line is an example of the typecasting issues that generic types aim to prevent.
- » The issue is that the 1.4.2 Collection API uses the Object class to store the Collection objects, which means that it cannot pick up type mismatches at compile time.
- » The same example with the generified Collections library is written as follows:

```
ArrayList<Integer> list = new ArrayList<Integer>();  
list.add(0, new Integer(42));  
int total = list.get(0).intValue();
```



# Collections Framework

---

- » The user of a generified API has to simply declare the type used at compile time using the <> notation.
- » No casts are needed and in this example trying to add a String object to an Integer typed collection would be caught at compile time.
- » Generic types therefore enable an API designer to provide common functionality that can be used with multiple data types and which also can be checked for type safety at compile time.
- » Designing your own Generic APIs is a little more complex than simply using them. To get started look at the `java.util.Collection` source and also the API guide.

# Collections Framework

---

- Autoboxing / Auto-Unboxing of Primitive Types
  - » Converting between primitive types, like int, boolean, and their equivalent Object-based counterparts like Integer and Boolean, can require unnecessary amounts of extra coding.
  - » Especially if the conversion is only needed for a method call to the Collections API, for example.
  - » The autoboxing and auto-unboxing of Java primitives produces code that is more concise and easier to follow.

# Collections Framework

---

- » J2SE 5.0 leaves the conversion required to transition to an Integer and back to the compiler.

- » Before

```
ArrayList<Integer> list = new ArrayList<Integer>();  
list.add(0, new Integer(42));  
int total = (list.get(0)).intValue();
```

- » After

```
ArrayList<Integer> list = new ArrayList<Integer>();  
list.add(0, 42);  
int total = list.get(0);
```

# Lab Assignment Seven

---

# GUI Programming

---

- See Chapter 11 of Textbook
- GUI Components

# Lab Assignment Eight

---

# Multithreaded Programming

- » A *process* is an instance of a program being executed.
  - In a *multitasking* environment, multiple processes can be running *concurrently*.
    - Actually, at a given instance in time, only one process is running on a single CPU. Time slicing between processes makes it appear as if they are all running at the same time.
- » A *thread* is a “lightweight process.”
  - Its like a process, but it doesn't have all the *overhead* that a process has.

# Processes vs. Threads

---

- » Both processes and threads have their own independent CPU state.
  - i.e. their own processor stack, instruction pointer, and CPU register values.
- » Multiple threads can share the same memory address space (i.e. share the same variables).
- » Processes, in general, do not share their address space with other processes.



# Threads and Java

---

- » Java has language level support for threads.
- » In Java it is easy to create multiple independent threads of execution.
  - Any class that is a subclass of `java.lang.Thread` or implements the `java.lang.Runnable` interface can be used to create threads.

# java.lang.Runnable

- » To create a thread, you instantiate the `java.lang.Thread` class.
  - One of `Thread`'s constructors takes objects that implement the `Runnable` interface.
  - The `Runnable` interface contains a single method called `run()`:

```
public interface Runnable {  
    public void run();  
}
```

# Creating and starting a thread

```
class Fred implements Runnable {  
    //..  
    public void run() {  
        //..  
    }  
  
    public static void main(String args[]) {  
        Thread t = new Thread(new Fred());  
        t.start();  
    }  
}
```

# start() and run()

- » To create a thread, create an instance of the Thread class.
- » To start a thread, call the start() method on the thread.
  - When the thread starts, the run() method is invoked.
- » The thread terminates when the run() method terminates.

# Thread priority and daemons

---

- » Every thread has a *priority*.
  - Threads with *higher priority* are executed in *preference* to threads with *lower priority*.
- » A thread may be marked as a *daemon*.
  - Daemon threads are *background threads* that are not expected to *exit*.
- » A *new thread* has its priority initially set equal to the priority of the *creating thread*.
  - The new thread is a daemon thread if and only if the creating thread is a daemon thread.

# JVM and threads

---

- » When the JVM starts up there is usually a single non-daemon thread initially.
  - This thread calls the `main()` method of an application.
- » The JVM continues to execute threads until either
  - The `exit()` method of the class `Runtime` is called,
  - or all non-daemon threads have died.

# Creating a new thread

## Method 1

---

- » Create a subclass of `java.lang.Thread`
  - This subclass should *override* the `run()` method.
- » An instance of this subclass can be created and started.

```
class PrimeThread extends Thread {
    long minPrime;
    PrimeThread(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime
        . . .
    }
}
```

# Method 1, continued...

---

- » The following code creates a thread and starts it running
  - The thread executes until its `run()` method terminates or it is hit over the head by calling the thread's `stop()` method.

```
PrimeThread p = new PrimeThread(143);  
p.start();
```



# Creating a new thread

## Method 2

---

- » The other way to create a thread is to declare a class that implements the `Runnable` interface.
- » An instance of the class can then be allocated, passed as an argument when instantiating the `Thread` class, and started.

# Method 2 continued...

---

```
class PrimeRun implements Runnable {
    long minPrime;
    PrimeRun(long minPrime) {
        this.minPrime = minPrime;
    }
    public void run() {
        // compute primes larger than minPrime
        . . .
    }
}
. . .
PrimeRun p = new PrimeRun(143);
new Thread(p).start();
```

# The death of a thread

---

- » Threads can be killed by calling their `stop()` method, but this is not a good idea in general.
- » Allow for a thread to *die naturally* by having its `run()` method return.
  - This often done by altering some variable that causes a while loop to terminate in the `run()` method.

# Simple Animation Applet

---

- When a program creates a thread, it can pass the *Thread* class constructor function an object whose statements will be executed.
- Using the *Thread* class *start* function, the program can start a thread's execution.
- The *start* function, in turn, will call the *Thread* objects *run* function.
- Rule of Thumb:
  - If an application or applet performs a time-consuming task, it should create and use its own thread of execution to perform that task.

# Simple Animation Applet

---

```
/* Threads example - this applet can serve as  
a template for most animation applets. It scrolls  
the message "Hello World!" across the screen */
```

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

```
public class HelloDes extends Applet implements  
    Runnable  
{  
    Thread HelloThread = null;  
    String Message = "Hello World!";
```

# Simple Animation Applet

---

```
Font font = new Font("TimesRoman",  
Font.BOLD, 24);  
int x, y;
```

```
// Once off initialization code
```

```
public void init()  
{  
    x = size().width;           // Display width  
    y = size().height / 2;     // Display height  
}
```

# Simple Animation Applet

---

// Create and start the animating thread.

```
public void start()
{
    if (HelloThread == null)
    {
        HelloThread = new Thread(this);
        HelloThread.start();
    }
}
```

# Simple Animation Applet

---

// Specify actual processing the thread will perform

```
public void run()
{
    while (true)
    {
        x = x-5;
        if (x == 0)
            x = size().width;

        repaint();
    }
}
```



# Simple Animation Applet

---

```
        try
        {
            HelloThread.sleep(500);
        }
        catch (InterruptedException e) {}
    }

    public void paint(Graphics g)
    {
        g.setFont(font);
        g.drawString(Message, x, y);
    }
}
```

# Synchronization

---

- » Multi-threaded coding requires special care
  - (also more difficult to debug)
- » You want to prevent multiple threads from altering the state of an object at the same time.
  - Sections of code that should not be executed simultaneously are called *critical sections*.
  - You want *mutual exclusion* of concurrent threads in *critical sections* of code.
  - This is done in Java using *synchronized methods* or *synchronized statements*.

# The synchronized statement

synchronized (expression) statement

- » `expression`: must resolve to an object or an array.
- » `statement`: the critical section, usually a statement block (i.e. surrounded by { and }).
- » A synchronized statement attempts to acquire a *lock* for the object or array specified by the `expression`.
  - Statement is not executed until the lock is obtained.

# More on synchronized

---

- » You do not have to use the synchronized statement unless multiple threads share data.

```
public static void sortArray(int[] a) {  
    synchronized (a) {  
        // sort the array here  
    }  
}
```

# Synchronized methods

---

- » The `synchronized` keyword is most often used as a *method modifier* in Java.
  - Indicates that the *entire method* is a *critical section*.
  - *Static synchronized methods*: Java obtains a lock for the *class*.
  - *Instance methods*: Java obtains an exclusive lock for the *class instance*.

```
public synchronized void sort() {  
    // whole method a critical section  
}
```

# Lab Assignment Nine

---

# Deadlocks

---

- » Using synchronization can actually cause problems.
  - Only the thread that holds the lock can execute the critical section.
  - When more than one lock is used a situation called *deadlock* can occur.
  - This happens when two or more threads are all waiting to acquire a lock that is held by one of the other waiting threads.
    - Each thread waiting for a lock will never release the locks they currently hold (impasse).

# Dining Philosophers Problem

- » The story goes like this:
  - Five philosophers are sitting at a round table.
  - In front of each philosopher is a bowl of rice.
  - Between each pair of philosophers is one chopstick.
  - Before an individual philosopher can take a bite of rice he must have two chopsticks one taken from the left, and one taken from the right.
  - The philosophers must find some way to share chopsticks such that they all get to eat.



# Dining Philosophers Problem

---

- » The algorithm works as follows:
  - The philosopher always reaches for the chopstick on his right first.
    - If the chopstick is there, he takes it and raises his right hand.
    - Next, he tries for the left chopstick.
    - If the chopstick is available, he picks it up and raises his other hand.
  - Now that the philosopher has both chopsticks, he takes a bite of rice and then puts both chopsticks down, allowing either of his two neighbours to get the chopsticks.

# Dining Philosophers Problem

---

- The philosopher then starts all over again by trying for the right chopstick.
- Between each attempt to grab a chopstick, each philosopher pauses for a random period of time.
- » This algorithm always ends up in deadlock!
  - All the philosophers are frozen with their right hand in the air. Why?
    - Because each philosopher immediately has one chopstick and is waiting on a condition that cannot be satisfied ...
    - They are all waiting for the left chopstick, which is held by the philosopher to their left.

# Solution to Dining Philosophers Problem

---

- » For most Java programmers, the best choice is to prevent deadlock rather than to try and detect it.
  - Deadlock detection is complicated and the simplest approach to preventing deadlock is to impose ordering on the condition variables.
  - In the dining philosopher algorithm, there is no ordering imposed on the condition variables because the philosophers and the chopsticks are arranged in a circle.
  - All chopsticks are equal.

# Solution to Dining Philosophers Problem

---

- We can number the chopsticks 1 through 5 and insisting that the philosophers pick up the chopstick with the lower number first.
  - The philosopher who is sitting between chopsticks 1 and 2 and the philosopher who is sitting between chopsticks 1 and 5 must now reach for the same chopstick first (chopstick 1) rather than picking up the one on the right.
  - Whoever gets chopstick 1 first is now free to take another one.
  - Whoever doesn't get chopstick 1 must now wait for the first philosopher to release it.
- Deadlock is not possible ...

# Producer Consumer Problem

---

- The Producer generates an integer between 0 and 9 (inclusive), stores it in an IntBuffer object, and prints the generated number.
- To make the synchronization problem more interesting, the Producer sleeps for a random amount of time between 0 and 100 mS before repeating the number generating cycle:
- The Consumer, being ravenous, consumes all integers from the IntBuffer (the exact same object into which the Producer put the integers in the first place) as quickly as they become available.

# Producer Consumer Problem

---

```
public class Producer extends Thread {  
    private IntBuffer cubbyhole;  
    private int number;  
  
    public Producer(IntBuffer c, int number) {  
        cubbyhole = c;  
        this.number = number;  
    }  
}
```

# Producer Consumer Problem

---

```
public void run() {  
    for (int i = 0; i < 10; i++) {  
        cubbyhole.put(i);  
        System.out.println("Producer #" + this.number  
            + " put: " + i);  
        try {  
            sleep((int)(Math.random() * 100));  
        } catch (InterruptedException e) { }  
    }  
}
```

# Producer Consumer Problem

---

```
public class Consumer extends Thread {  
    private IntBuffer cubbyhole;  
    private int number;  
  
    public Consumer(IntBuffer c, int number) {  
        cubbyhole = c;  
        this.number = number;  
    }  
}
```



# Producer Consumer Problem

---

```
public void run() {  
    int value = 0;  
    for (int i = 0; i < 10; i++) {  
        value = cubbyhole.get();  
        System.out.println("Consumer #" +  
            this.number + " got: " + value);  
    }  
}  
}
```

# Producer Consumer Problem

---

- The Producer and Consumer in this example share data through a common IntBuffer object.
- Note that neither the Producer nor the Consumer makes any effort whatsoever to ensure that the Consumer is getting each value produced once and only once.
- The synchronization between these two threads actually occurs at a lower level, within the get() and put() methods of the IntBuffer object.

# Producer Consumer Problem

---

- Race conditions arise from multiple, asynchronous executing threads trying to access a single object at the same time and getting the wrong result.
- Race conditions in the producer/consumer example are prevented by having the storage of a new integer into the IntBuffer by the Producer be synchronized with the retrieval of an integer from the IntBuffer by the Consumer.
- The Consumer must consume each integer produced by the Producer exactly once.

# wait() and notify()

- » JAVA monitors are re-entrant:
  - Once a thread obtains a monitor (lock) it can release it again and wait, using wait(), pending the completion of some other event or action by another thread.
  - Useful in situations where separate threads are supplying and consuming data or events.
  - Can only be executed within synchronized code or methods and helps prevent deadlock and uncoordinated access to shared data ...
  - IntBuffer implementation uses these methods.

# IntBuffer Implementation

---

– Here's the code skeleton for the IntBuffer class:

```
public class IntBuffer {  
    private int contents;  
    private boolean available = false;  
  
    public synchronized int get() {  
        ...  
    }  
  
    public synchronized void put(int value) {  
        ...  
    }  
}
```

# IntBuffer Implementation

---

- Note that the method declarations for both put and get contain the synchronized keyword.
- Hence, the system associates a unique lock with every instance of IntBuffer (including the one shared by the Producer and the Consumer).
- Whenever control enters a synchronized method, the thread that called the method locks the object whose method has been called.
- Other threads cannot call a synchronized method on the same object until the object is unlocked.

# IntBuffer Implementation

---

- When the Producer calls IntBuffer's put method, it locks the IntBuffer, thereby preventing the Consumer from calling the IntBuffer's get method:

```
public synchronized void put(int value) {  
    // IntBuffer locked by the Producer  
  
    ..  
    // IntBuffer unlocked by the Producer  
}
```

- When the put method returns, the Producer unlocks the IntBuffer.

# IntBuffer Implementation

---

- Similarly, when the Consumer calls IntBuffer's get method, it locks the IntBuffer, thereby preventing the Producer from calling put:

```
public synchronized int get() {  
    // IntBuffer locked by the Consumer  
    ...  
    // IntBuffer unlocked by the Consumer  
}
```



# IntBuffer Implementation

---

- The IntBuffer stores its value in a private member variable called *contents*.
- IntBuffer has another private member variable, *available*, that is a boolean.
- *available* is true when the value has just been put but not yet gotten and is false when the value has been gotten but not yet put.
- There follows one possible implementation for the put and get methods:

# IntBuffer Implementation

---

```
public synchronized int get() { // won't work!
    if (available == true) {
        available = false;
        return contents;
    }
}

public synchronized void put(int value) { // won't work!
    if (available == false) {
        available = true;
        contents = value;
    }
}
```

# IntBuffer Implementation

---

- As implemented, these two methods won't work. Look at the *get()* method...
- What happens if the Producer hasn't put anything in the IntBuffer and *available* isn't true? *get()* does nothing.
- Similarly, if the Producer calls *put()* before the Consumer got the value, *put()* does nothing.
- You really want the Consumer to wait until the Producer puts something in the IntBuffer and the Producer should then notify the Consumer when it's done so.

# IntBuffer Implementation

---

- Similarly, the Producer must wait until the Consumer takes a value (and notifies the Producer of its activities) before replacing it with a new value.
- The two threads must co-ordinate more fully and can use Object's `wait()` and `notifyAll()` methods to do so.
- There follows the new implementations of *get()* and *put()* that wait on and notify each other of their activities:

# IntBuffer Implementation

---

```
public synchronized int get() {  
    while (available == false) {  
        try {  
            // wait for Producer to put value  
            wait();  
        } catch (InterruptedException e) {  
        }  
    }  
    available = false;  
    // notify Producer that value has been retrieved  
    notifyAll();  
    return contents;  
}
```

# IntBuffer Implementation

---

```
public synchronized void put(int value) {  
    while (available == true) {  
        try {  
            // wait for Consumer to get value  
            wait();  
        } catch (InterruptedException e) {  
        }  
    }  
    contents = value;  
    available = true;  
    // notify Consumer that value has been set  
    notifyAll();  
}
```

# IntBuffer Implementation

---

- The code in the `get()` method loops until the Producer has produced a new value - each time through the loop, `get()` calls the `wait()` method.
- The `wait()` method relinquishes the lock held by the Consumer on the `IntBuffer` (thereby allowing the Producer to get the lock and update the `IntBuffer`) and then waits for notification from the Producer.
- When the Producer puts something in the `IntBuffer`, it notifies the Consumer by calling `notifyAll()`.

# IntBuffer Implementation

---

- The Consumer then comes out of the wait state, available is now true, the loop exits, and the get() method returns the value in the IntBuffer.
- The put() method works in a similar fashion, waiting for the Consumer thread to consume the current value before allowing the Producer to produce a new one.
- The notifyAll() method wakes up all threads waiting on the object in question (in this case, the IntBuffer).



# IntBuffer Implementation

---

- The awakened threads compete for the lock - one thread gets it, and the others go back to waiting.
- The `java.lang.Object` class also defines the `notify()` method, which arbitrarily wakes up only one of the threads waiting on this object.
- The `Object` class also contains two other versions of the wait method:
  - `wait(long timeout)` waits for notification or until the timeout period (in mS) has elapsed.
  - `wait(long timeout, int nanos)` waits for notification or until timeout mS plus nS nanoseconds have elapsed.

# Concurrency Utilities

---

- The Java 2 platform now includes a new package of concurrency utilities.
  - » These are classes which are designed to be used as building blocks in building concurrent classes or applications.
  - » The Concurrency Utilities include a high-performance, flexible thread pool; a framework for asynchronous execution of tasks; a host of collection classes optimized for concurrent access; synchronization utilities such as counting semaphores; atomic variables; locks; and condition variables.

# Concurrency Utilities

---

- Using the Concurrency Utilities, instead of developing components such as thread pools yourself, offers a number of advantages:
  - » Reduced programming effort. It is far easier to use a standard class than to develop it yourself.
  - » Increased performance. The implementations in the Concurrency Utilities were developed and peer-reviewed by concurrency and performance experts; these implementations are likely to be faster and more scalable than a typical implementation, even by a skilled developer.
  - » Increased reliability. Developing concurrent classes is difficult -- the low-level concurrency primitives provided by the Java language (`synchronized`, `volatile`, `wait()`, `notify()`, and `notifyAll()`) are difficult to use correctly, and errors using these facilities can be difficult to detect and debug.

# Concurrency Utilities

---

- » By using standardized, extensively tested concurrency building blocks, many potential sources of threading hazards such as deadlock, starvation, race conditions, or excessive context switching are eliminated.
- » Improved maintainability. Programs which use standard library classes are easier to understand and maintain than those which rely on complicated, homegrown classes.
- » Increased productivity. Developers are likely to already understand the standard library classes, so there is no need to learn the API and behavior of ad-hoc concurrent components.

# Thread Pools

---

- Thread Pools

- » A thread pool is a managed collection of threads that are available to perform tasks. Thread pools usually provide:
- » Improved performance when executing large numbers of tasks due to reduced per-task invocation overhead.
- » A means of bounding the resources, including threads, consumed when executing a collection of tasks.
- » In addition, thread pools relieve you from having to manage the life cycle of threads. They allow to take advantage of threading, but focus on the tasks that you want the threads to perform, instead of the thread mechanics.

# Thread Pools

---

- » To use thread pools, you instantiate an implementation of the `ExecutorService` interface and hand it a set of tasks.
- » The choices of configurable thread pool implementations are `ThreadPoolExecutor` and `ScheduledThreadPoolExecutor`.
- » These implementations allow you to set the core and maximum pool size, the type of data structure used to hold the tasks, how to handle rejected tasks, and how to create and terminate threads.
- » However, we recommend that you use the more convenient factory methods of the `Executors` class listed in the following table. These methods preconfigure settings for the most common usage scenarios.

# Thread Pools

---

## » Factory Methods in the Executors Class

<u>Method</u>	<u>Description</u>
<code>newFixedThreadPool(int)</code>	Creates a fixed size thread pool.
<code>newCachedThreadPool</code>	Creates unbounded thread pool, with automatic thread reclamation.
<code>newSingleThreadExecutor</code>	Creates a single background thread.

# Thread Pools

---

```
public class WorkerThread implements Runnable {
    private int workerNumber;

    WorkerThread(int number) {
        workerNumber = number;
    }

    public void run() {
        for (int i=0;i<=100;i+=20) {
            // Perform some work ...
            System.out.println("Worker number: "
                + workerNumber
                + ", percent complete: " + i );
            try {
                Thread.sleep((int)(Math.random() * 1000));
            } catch (InterruptedException e) {
            }
        }
    }
}
```



# Thread Pools

---

```
import java.util.concurrent.*;
public class ThreadPoolTest {
    public static void main(String[] args) {
        int numWorkers = Integer.parseInt(args[0]);
        int threadPoolSize = Integer.parseInt(args[1]);

        ExecutorService tpes =
            Executors.newFixedThreadPool(threadPoolSize);

        WorkerThread[] workers = new WorkerThread[numWorkers];
        for (int i = 0; i < numWorkers; i++) {
            workers[i] = new WorkerThread(i);
            tpes.execute(workers[i]);
        }
        tpes.shutdown();
    }
}
```

# Lab Assignment Ten

---

# Sockets API

---

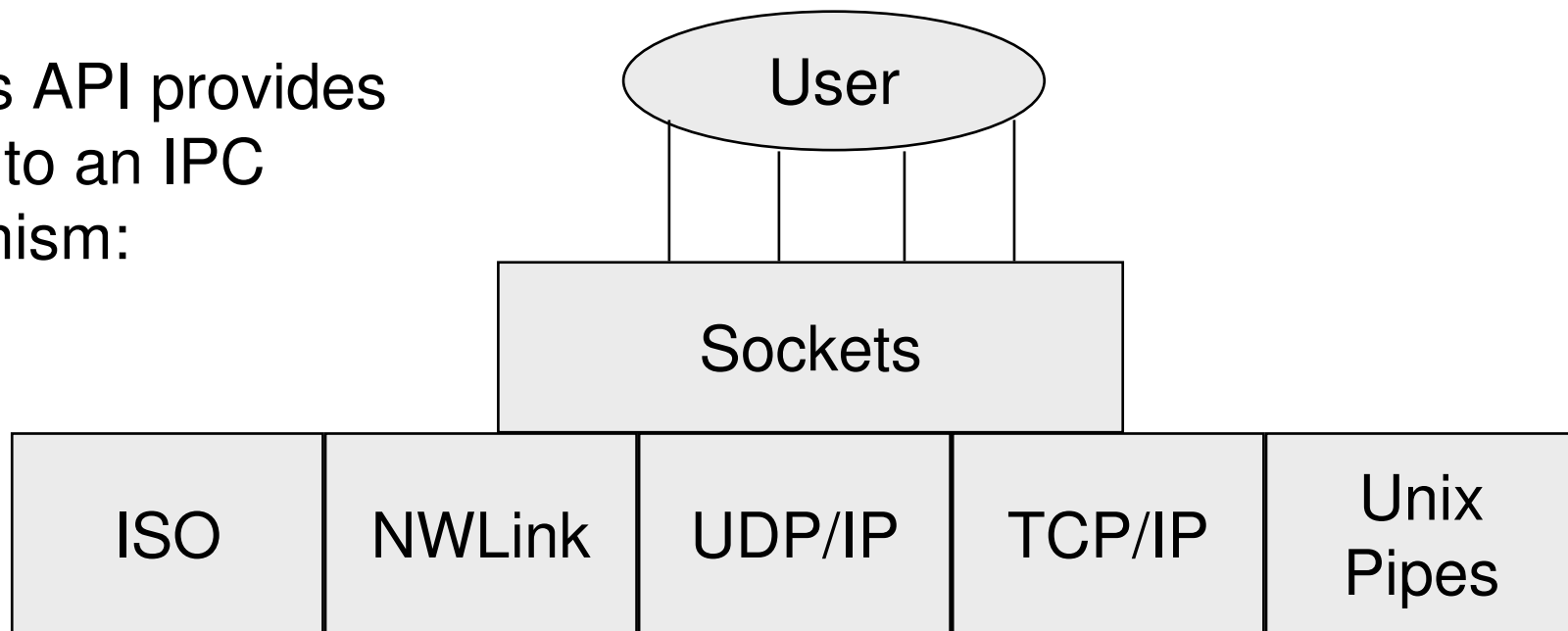
- Sockets Abstraction

- Sockets were developed as a BSD Unix system abstraction to allow users to write programs that utilise underlying comms protocols.
- Unrelated processes can communicate with each other (across the network).
- *Endpoint* of communication to which a *name* may be bound.
- Allow developer to write programs that are *largely* independent of lower protocols.

# Sockets API

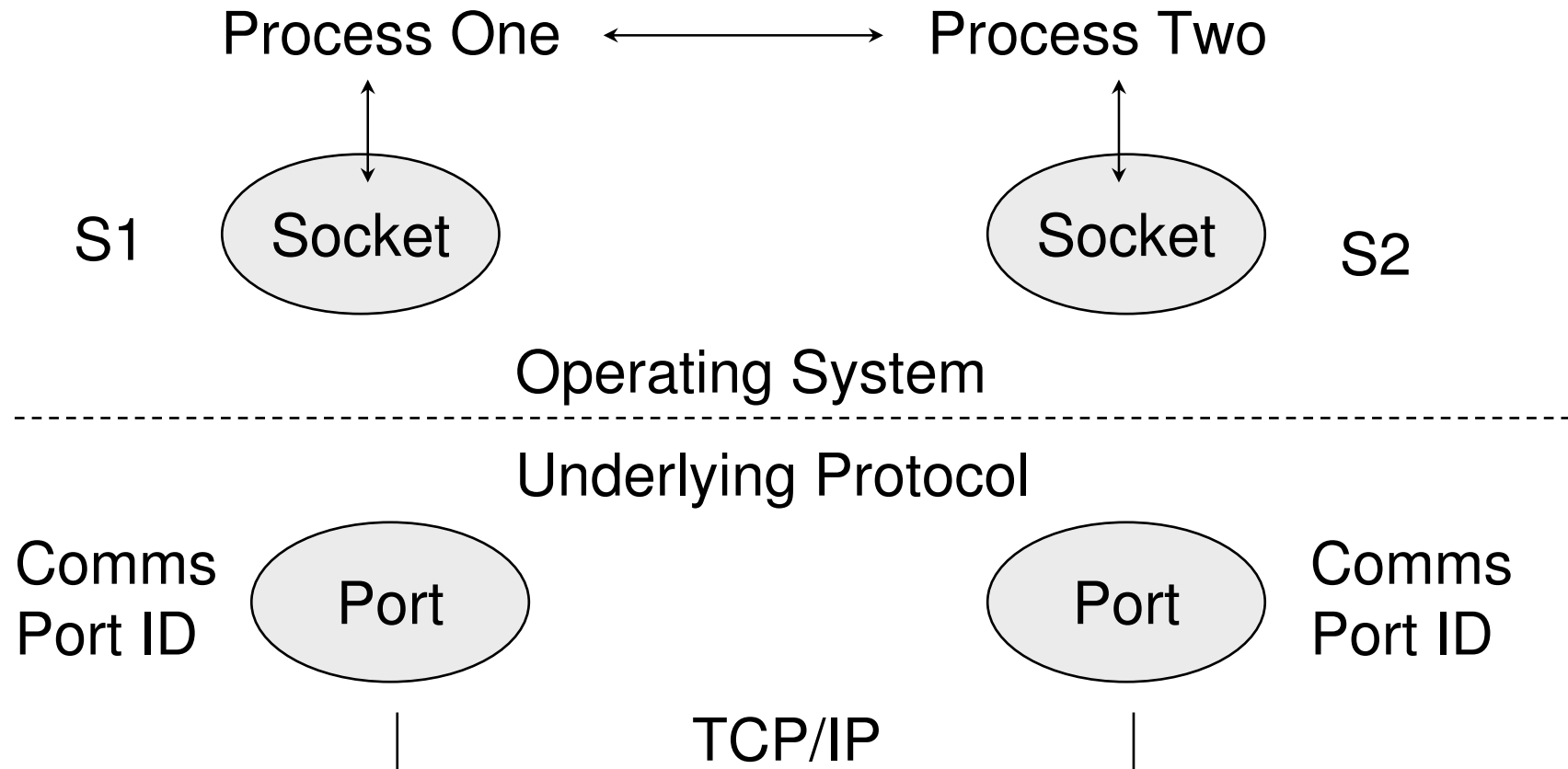
---

Sockets API provides access to an IPC mechanism:



# Sockets API

---



# Sockets API

---

- Two Sockets are required for end-to-end communication.
- Biased towards Client / Server Model.
- Sockets must be:
  - **Created** at both ends (specifying protocol type).
  - **Bound** to a Local Address (Named).
  - Optionally **Connected** to a (remote) socket.
- A socket identifier is returned:
  - This is similar to a file ID and they share many operations.
  - Passed to forked child processes.

# Sockets API

---

- Types of sockets

- In BSD Unix, and the systems derived from it, socket functions are part of the OS itself.
- As usage increased, other vendors decided to add a Sockets API to their systems.
- Often this was in the form of a *sockets library* that provides the Sockets API layered above an underlying set of native *system calls*.
- In practice, Socket libraries are seldom perfect ! Minor differences sometimes occur e.g. in the way errors are handled.

# JAVA Sockets

---

- Networking package is **java.net**
  - » Socket-based communications
    - Applications view networking as streams of data
    - Connection-based protocol
    - Uses TCP (Transmission Control Protocol)
  - » Packet-based communications
    - Individual packets transmitted
    - Connectionless service
    - Uses UDP (User Datagram Protocol)



# JAVA Sockets

---

- Sockets in Java

- A socket is one end-point of a two-way communication link between two programs running on the network.
- Socket classes are used to represent the connection between a client program and a server program.
- The java.net package provides two classes - *Socket* and *ServerSocket*:
  - These implement the client side of the connection and the server side of the connection, respectively.

# JAVA Sockets

---

```
// Network server that detects the presence of  
// dogs on the Internet !
```

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

```
class server {  
    public static void main(String a[]) throws  
        IOException {  
        int timeoutsecs = 600;  
        int port = 4444;  
        Socket sock;
```

# JAVA Sockets

---

```
String query = "If you met me would you say  
hello, or bark ?";
```

```
ServerSocket servsock = new  
    ServerSocket(port, timeoutsecs);
```

```
while (true) {  
    // wait for the next client connection  
    sock=servsock.accept();
```

```
    // Get I/O streams from the socket  
    PrintStream out = new PrintStream(  
        sock.getOutputStream() );
```

# JAVA Sockets

---

```
DataInputStream in = new DataInputStream(  
sock.getInputStream() );
```

```
// Send our query  
out.println(query);  
out.flush();
```

```
// get the reply  
String reply = in.readLine();  
if (reply.indexOf("bark") > -1)  
System.out.println("On the Internet I  
know this is a DOG!");
```

# JAVA Sockets

---

```
else System.out.println("Probably a person  
or an AI experiment");
```

```
    // Close this connection, (not the overall      //  
    server socket)  
    sock.close();  
    }  
}  
}
```

```
// However, no one knows you're a dog...  
// unless you tell them (we need a client) !
```

# JAVA Sockets

---

```
// This is the Client ...
```

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

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

```
class dog {
```

```
    public static void main(String a[]) throws
```

```
        IOException {
```

```
            Socket sock;
```

```
            DataInputStream dis;
```

```
            PrintStream dat;
```

# JAVA Sockets

---

```
// Open our connection to dcham, at port 4444
// If you try this on your system, insert your system
// in place of "dcham" - "dcham.nuigalway.ie" is my
// system name.
sock = new Socket("dcham",4444);

// Get I/O streams from the socket
dis = new DataInputStream( sock.getInputStream() );
dat = new PrintStream( sock.getOutputStream() );

String fromServer = dis.readLine();
```

# JAVA Sockets

---

```
        System.out.println("Got this from server:" +  
                            fromServer);  
        // assume that we expected the question,  
so  
        // just go ahead and answer  
  
        dat.println("I would bark !");  
        dat.flush();  
  
        sock.close();  
    }  
}
```



# Connectionless Client/Server

---

- » Connectionless transmission with datagrams
  - No connection maintained with other computer
  - Break message into equal sized pieces and send as packets
  - Message arrive in order, out of order or not at all
  - Receiver puts messages in order and reads them

```

1  // Fig. 17.6: Server.java
2  // Set up a Server that will receive packets from a
3  // client and send packets to a client.
4
5  // Java core packages
6  import java.io.*;
7  import java.net.*;
8  import java.awt.*;
9  import java.awt.event.*;
10
11 // Java extension packages
12 import javax.swing.*;
13
14 public class Server extends JFrame {
15     private JTextArea displayArea;
16     private DatagramPacket sendPacket, receivePacket;
17     private DatagramSocket socket;
18
19     // set up GUI and DatagramSocket
20     public Server() ←
21     {
22         super( "Server" );
23
24         displayArea = new JTextArea();
25         getContentPane().add( new JScrollPane( displayArea ),
26             BorderLayout.CENTER );
27         setSize( 400, 300 );
28         setVisible( true );
29
30         // create DatagramSocket for sending and receiving packets
31         try {
32             socket = new DatagramSocket( 5000 ); ←
33         }
34

```

Constructor creates  
GUI

Create  
DatagramSocket at  
port 5000

```

35     // process problems creating DatagramSocket
36     catch( SocketException socketException ) {
37         socketException.printStackTrace();
38         System.exit( 1 );
39     }
40
41 } // end Server constructor
42
43 // wait for packets to arrive, then display data and echo
44 // packet to client
45 public void waitForPackets()
46 {
47     // loop forever
48     while ( true ) {
49
50         // receive packet, display contents, echo to client
51         try {
52
53             // set up packet
54             byte data[] = new byte[ 100 ];
55             receivePacket =
56                 new DatagramPacket( data, data.length );
57
58             // wait for packet
59             socket.receive( receivePacket );
60
61             // process packet
62             displayPacket();
63
64             // echo information from packet back to client
65             sendPacketToClient();
66         }
67     }

```

Method  
**waitForPackets**  
uses an infinite loop to  
wait for packets to  
arrive

Create a  
**DatagramPacket** to  
store received  
information

Method **receive**  
blocks until a packet is  
received

```

68         // process problems manipulating packet
69         catch( IOException ioException ) {
70             displayArea.append( ioException.toString() + "\n" );
71             ioException.printStackTrace();
72         }
73     } // end while
74 } // end method waitForPackets
75
76 // display packet contents
77 private void displayPacket()
78 {
79     displayArea.append( "\nPacket received:" +
80         "\nFrom host: " + receivePacket.getAddress() +
81         "\nHost port: " + receivePacket.getPort() +
82         "\nLength: " + receivePacket.getLength() +
83         "\nContaining:\n\t" +
84         new String( receivePacket.getData(), 0,
85             receivePacket.getLength() ) );
86 }
87
88 // echo packet to client
89 private void sendPacketToClient() throws IOException
90 {
91     displayArea.append( "\n\nEcho data to client..." );
92
93     // create packet to send
94     sendPacket = new DatagramPacket( receivePacket.getData(),
95         receivePacket.getLength(), receivePacket.getAddress(),
96         receivePacket.getPort() );
97
98     // send packet
99     socket.send( sendPacket );
100
101

```

Method **displayPacket** appends packet's contents to **displayArea**

Method **getAddress** returns name of computer that sent packet

Method **getPort** returns the port the packet came through

Method **getLength** returns the length of the message sent

Method **getData** returns a byte array containing the sent data

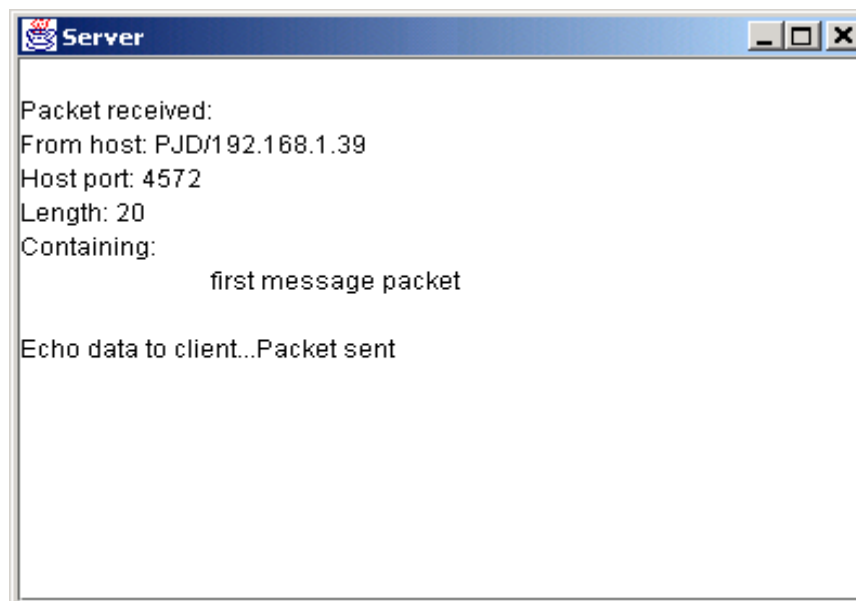
Method **send** sends the pack over the network

```

102
103     displayArea.append( "Packet sent\n" );
104     displayArea.setCaretPosition(
105         displayArea.getText().length() );
106 }
107
108 // execute application
109 public static void main( String args[] )
110 {
111     Server application = new Server();
112
113     application.setDefaultCloseOperation(
114         JFrame.EXIT_ON_CLOSE );
115
116     application.waitForPackets();
117 }
118
119 } // end class Server

```

Method **main** creates a new server and waits for packets



```

1  // Fig. 17.7: Client.java
2  // Set up a Client that will send packets to a
3  // server and receive packets from a server.
4
5  // Java core packages
6  import java.io.*;
7  import java.net.*;
8  import java.awt.*;
9  import java.awt.event.*;
10
11 // Java extension packages
12 import javax.swing.*;
13
14 public class Client extends JFrame {
15     private JTextField enterField;
16     private JTextArea displayArea;
17     private DatagramPacket sendPacket, receivePacket;
18     private DatagramSocket socket;
19
20     // set up GUI and DatagramSocket
21     public Client() ←
22     {
23         super( "Client" );
24
25         Container container = getContentPane();
26
27         enterField = new JTextField( "Type message here" );
28

```

Constructor sets up  
GUI and  
**DatagramSocket**  
object

```

29     enterField.addActionListener(
30
31         new ActionListener() {
32
33             // create and send a packet
34             public void actionPerformed((ActionEvent event)
35             {
36                 // create and send packet
37                 try {
38                     displayArea.append(
39                         "\nSending packet containing: " +
40                         event.getActionCommand() + "\n" );
41
42                     // get message from textfield and convert to
43                     // array of bytes
44                     String message = event.getActionCommand();
45                     byte data[] = message.getBytes();
46
47                     // create sendPacket
48                     sendPacket = new DatagramPacket(
49                         data, data.length,
50                         InetAddress.getLocalHost(), 5000 );
51
52                     // send packet
53                     socket.send( sendPacket );
54
55                     displayArea.append( "Packet sent\n" );
56                     displayArea.setCaretPosition(
57                         displayArea.getText().length() );
58                 }
59

```

Method  
**actionPerformed**  
converts a **String** to a  
**byte** array to be sent as  
a datagram

Convert the **String** to  
a **byte** array

Create the  
**DatagramPacket** to  
send

Send the packet with  
method **send**

```

60         // process problems creating or sending packet
61         catch ( IOException ioException ) {
62             displayArea.append(
63                 ioException.toString() + "\n" );
64             ioException.printStackTrace();
65         }
66
67     } // end actionPerformed
68
69 } // end anonymous inner class
70
71 ); // end call to addActionListener
72
73 container.add( enterField, BorderLayout.NORTH );
74
75 displayArea = new JTextArea();
76 container.add( new JScrollPane( displayArea ),
77     BorderLayout.CENTER );
78
79 setSize( 400, 300 );
80 setVisible( true );
81
82 // create DatagramSocket for sending and receiving packets
83 try {
84     socket = new DatagramSocket();
85 }
86
87 // catch problems creating DatagramSocket
88 catch( SocketException socketException ) {
89     socketException.printStackTrace();
90     System.exit( 1 );
91 }
92
93 } // end Client constructor
94

```

Create  
**DatagramSocket**  
for sending and  
receiving packets



```

95  // wait for packets to arrive from Server,
96  // then display packet contents
97  public void waitForPackets()
98  {
99      // loop forever
100     while ( true ) {
101
102         // receive packet and display contents
103         try {
104
105             // set up packet
106             byte data[] = new byte[ 100 ];
107             receivePacket =
108                 new DatagramPacket( data, data.length );
109
110             // wait for packet
111             socket.receive( receivePacket );
112
113             // display packet contents
114             displayPacket();
115         }
116
117         // process problems receiving or displaying packet
118         catch( IOException exception ) {
119             displayArea.append( exception.toString() + "\n" );
120             exception.printStackTrace();
121         }
122
123     } // end while
124
125 } // end method waitForPackets
126

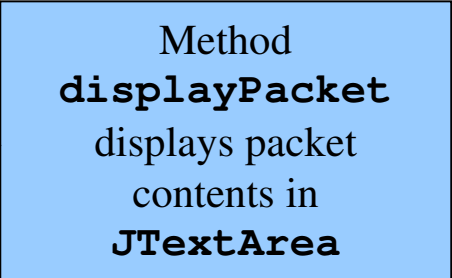
```

Method  
**waitForPackets**  
uses an infinite loop to  
wait for packets from  
server

Block until packet  
arrives

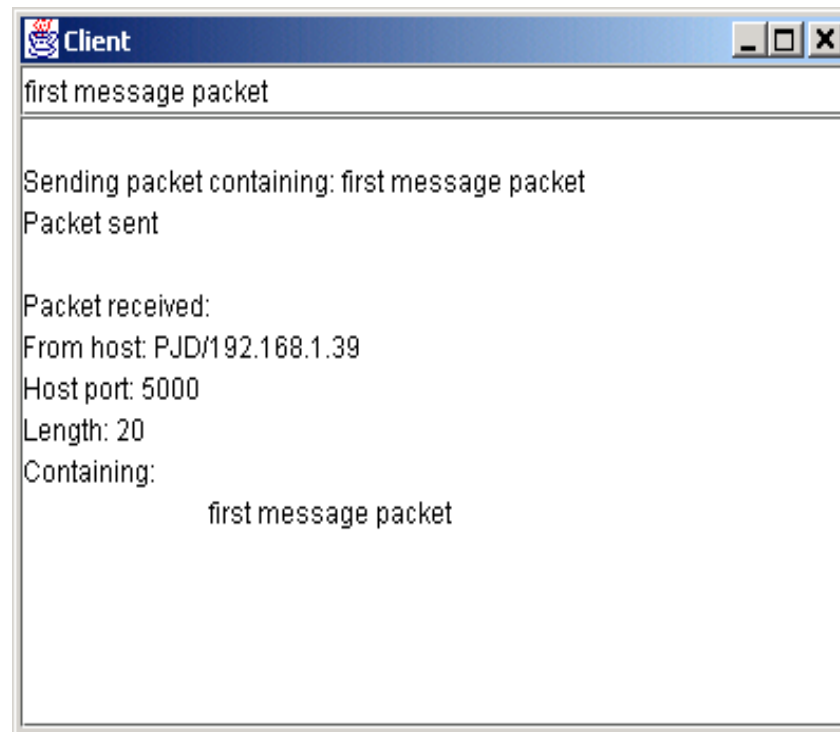
Display contents of  
packet

```
127 // display contents of receivePacket
128 private void displayPacket()
129 {
130     displayArea.append( "\nPacket received:" +
131         "\nFrom host: " + receivePacket.getAddress() +
132         "\nHost port: " + receivePacket.getPort() +
133         "\nLength: " + receivePacket.getLength() +
134         "\nContaining:\n\t" +
135         new String( receivePacket.getData(), 0,
136             receivePacket.getLength() ) );
137
138     displayArea.setCaretPosition(
139         displayArea.getText().length() );
140 }
141
142 // execute application
143 public static void main( String args[] )
144 {
145     Client application = new Client();
146
147     application.setDefaultCloseOperation(
148         JFrame.EXIT_ON_CLOSE );
149
150     application.waitForPackets();
151 }
152
153 } // end class Client
```



Method  
**displayPacket**  
displays packet  
contents in  
**JTextArea**

# Program Output



# Multithreaded Server Example

- Tic-Tac-Toe Server
- Multiple threads
  - Server uses one thread per player
    - Allow each player to play game independently

```

1  // Fig. 17.8: TicTacToeServer.java
2  // This class maintains a game of Tic-Tac-Toe for two
3  // client applets.
4
5  // Java core packages
6  import java.awt.*;
7  import java.awt.event.*;
8  import java.net.*;
9  import java.io.*;
10
11 // Java extension packages
12 import javax.swing.*;
13
14 public class TicTacToeServer extends JFrame {
15     private byte board[];
16     private JTextArea outputArea;
17     private Player players[];
18     private ServerSocket server;
19     private int currentPlayer;
20
21 // set up tic-tac-toe server and GUI that displays messages
22 public TicTacToeServer() ←
23 {
24     super( "Tic-Tac-Toe Server" );
25
26     board = new byte[ 9 ];
27     players = new Player[ 2 ];
28     currentPlayer = 0;
29
30 // set up ServerSocket
31 try {
32     server = new ServerSocket( 5000, 2 );
33 }
34

```

Constructor creates a  
**ServerSocket** and  
server GUI

```

35      // process problems creating ServerSocket
36      catch( IOException ioException ) {
37          ioException.printStackTrace();
38          System.exit( 1 );
39      }
40
41      // set up JTextArea to display messages during execution
42      outputArea = new JTextArea();
43      getContentPane().add( outputArea, BorderLayout.CENTER );
44      outputArea.setText( "Server awaiting connections\n" );
45
46      setSize( 300, 300 );
47      setVisible( true );
48  }
49
50  // wait for two connections so game can be played
51  public void execute()
52  {
53      // wait for each client to connect
54      for ( int i = 0; i < players.length; i++ ) {
55
56          // wait for connection, create Player, start thread
57          try {
58              players[ i ] = new Player( server.accept(), i );
59              players[ i ].start();
60          }
61
62          // process problems receiving connection from client
63          catch( IOException ioException ) {
64              ioException.printStackTrace();
65              System.exit( 1 );
66          }
67      }
68  }

```

Method **execute**  
waits for two  
connections to start  
game

Block while waiting for  
each player

Call **start** method to  
begin executing thread

```

69     // Player X is suspended until Player O connects.
70     // Resume player X now.
71     synchronized ( players[ 0 ] ) {
72         players[ 0 ].setSuspended( false );
73         players[ 0 ].notify();
74     }
75
76 } // end method execute
77
78 // display a message in outputArea
79 public void display( String message )
80 {
81     outputArea.append( message + "\n" );
82 }
83
84 // Determine if a move is valid.
85 // This method is synchronized because only one move can be
86 // made at a time.
87 public synchronized boolean validMove(
88     int location, int player )
89 {
90     boolean moveDone = false;
91
92     // while not current player, must wait for turn
93     while ( player != currentPlayer ) {
94
95         // wait for turn
96         try {
97             wait();
98         }
99
100        // catch wait interruptions
101        catch( InterruptedException interruptedException ) {
102            interruptedException.printStackTrace();
103        }

```

Method **validMove**  
allows only one move  
at a time

```

104     }
105
106     // if location not occupied, make move
107     if ( !isOccupied( location ) ) {
108
109         // set move in board array
110         board[ location ] =
111             ( byte ) ( currentPlayer == 0 ? 'X' : 'O' );
112
113         // change current player
114         currentPlayer = ( currentPlayer + 1 ) % 2;
115
116         // let new current player know that move occurred
117         players[ currentPlayer ].otherPlayerMoved( location );
118
119         // tell waiting player to continue
120         notify();
121
122         // tell player that made move that the move was valid
123         return true;
124     }
125
126     // tell player that made move that the move was not valid
127     else
128         return false;
129 }
130
131 // determine whether location is occupied
132 public boolean isOccupied( int location )
133 {
134     if ( board[ location ] == 'X' || board [ location ] == 'O' )
135         return true;
136     else
137         return false;
138 }

```

Place a mark on the  
board

Notify player a move  
occurred

Invoke method  
**notify** to tell waiting  
player to continue

Confirm valid move to  
player



```

139
140 // place code in this method to determine whether game over
141 public boolean gameOver()
142 {
143     return false;
144 }
145
146 // execute application
147 public static void main( String args[] )
148 {
149     TicTacToeServer application = new TicTacToeServer();
150
151     application.setDefaultCloseOperation(
152         JFrame.EXIT_ON_CLOSE );
153
154     application.execute();
155 }
156
157 // private inner class Player manages each Player as a thread
158 private class Player extends Thread {
159     private Socket connection;
160     private DataInputStream input;
161     private DataOutputStream output;
162     private int playerNumber;
163     private char mark;
164     protected boolean suspended = true;
165
166     // set up Player thread
167     public Player( Socket socket, int number )
168     {
169         playerNumber = number;
170
171         // specify player's mark
172         mark = ( playerNumber == 0 ? 'X' : 'O' );
173

```

Method main creates an instance of **TicTacToeServer** and calls method **execute**

The **Player** constructor receives a **Socket** object and gets its input and output streams

```

174         connection = socket;
175
176         // obtain streams from Socket
177         try {
178             input = new DataInputStream(
179                 connection.getInputStream() );
180             output = new DataOutputStream(
181                 connection.getOutputStream() );
182         }
183
184         // process problems getting streams
185         catch( IOException ioException ) {
186             ioException.printStackTrace();
187             System.exit( 1 );
188         }
189     }
190
191     // send message that other player moved; message contains
192     // a String followed by an int
193     public void otherPlayerMoved( int location )
194     {
195         // send message indicating move
196         try {
197             output.writeUTF( "Opponent moved" );
198             output.writeInt( location );
199         }
200
201         // process problems sending message
202         catch ( IOException ioException ) {
203             ioException.printStackTrace();
204         }
205     }
206

```

```

207 // control thread's execution
208 public void run()
209 {
210     // send client message indicating its mark (X or O),
211     // process messages from client
212     try {
213         display( "Player " + ( playerNumber == 0 ?
214             'X' : 'O' ) + " connected" );
215
216         // send player's mark
217         output.writeChar( mark );
218
219         // send message indicating connection
220         output.writeUTF( "Player " +
221             ( playerNumber == 0 ? "X connected\n" :
222                 "O connected, please wait\n" ) );
223
224         // if player X, wait for another player to arrive
225         if ( mark == 'X' ) {
226             output.writeUTF( "Waiting for another player" );
227
228             // wait for player O
229             try {
230                 synchronized( this ) {
231                     while ( suspended )
232                         wait();
233                 }
234             }
235
236             // process interruptions while waiting
237             catch ( InterruptedException exception ) {
238                 exception.printStackTrace();
239             }
240

```

Method **run** controls  
information sent and  
received by client

Tell client that client's  
connection is made

Send player's mark

Suspend each thread as  
it starts executing

```

241         // send message that other player connected and
242         // player X can make a move
243         output.writeUTF(
244             "Other player connected. Your move." );
245     }
246
247     // while game not over
248     while ( ! gameOver() ) {
249
250         // get move location from client
251         int location = input.readInt();
252
253         // check for valid move
254         if ( validMove( location, playerNumber ) ) {
255             display( "loc: " + location );
256             output.writeUTF( "Valid move." );
257         }
258         else
259             output.writeUTF( "Invalid move, try again" );
260     }
261
262     // close connection to client
263     connection.close();
264 }
265
266 // process problems communicating with client
267 catch( IOException ioException ) {
268     ioException.printStackTrace();
269     System.exit( 1 );
270 }
271 }
272

```

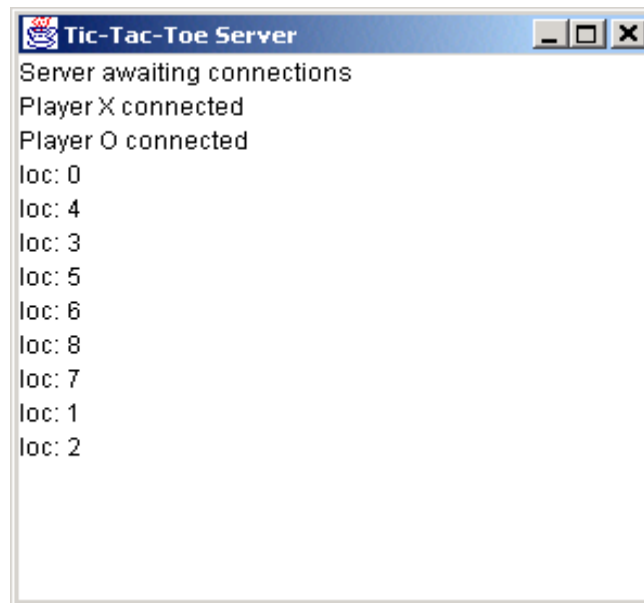
Loop until game is over

Read move location

Check if valid move

Send message to client

```
273      // set whether or not thread is suspended
274      public void setSuspended( boolean status )
275      {
276          suspended = status;
277      }
278
279  } // end class Player
280
281  } // end class TicTacToeServer
```



# OO Design Issues

---

- » When a system gets large it gets complex:
  - Large number of people working together must not hinder each other !
  - Changes can propagate ...
  - Change control contention ...
  - Parallel development can become impossible ..
- » The open/closed principle
  - This should guide decisions at all levels.
  - Classes/components should be open for extension but closed to modification ...

# Open / Closed Principle

---

– Open for modification example:

```
void paint()
{
    for (each shape)
        switch(shape.type)
        {
            case: Shape.CIRCLE:
                drawCircle((Circle) shape);
                break;
            case: Shape.RECTANGLE:
                drawRectangle((Rectangle) shape);
                break;
        }
}
```

# Open / Closed Principle

---

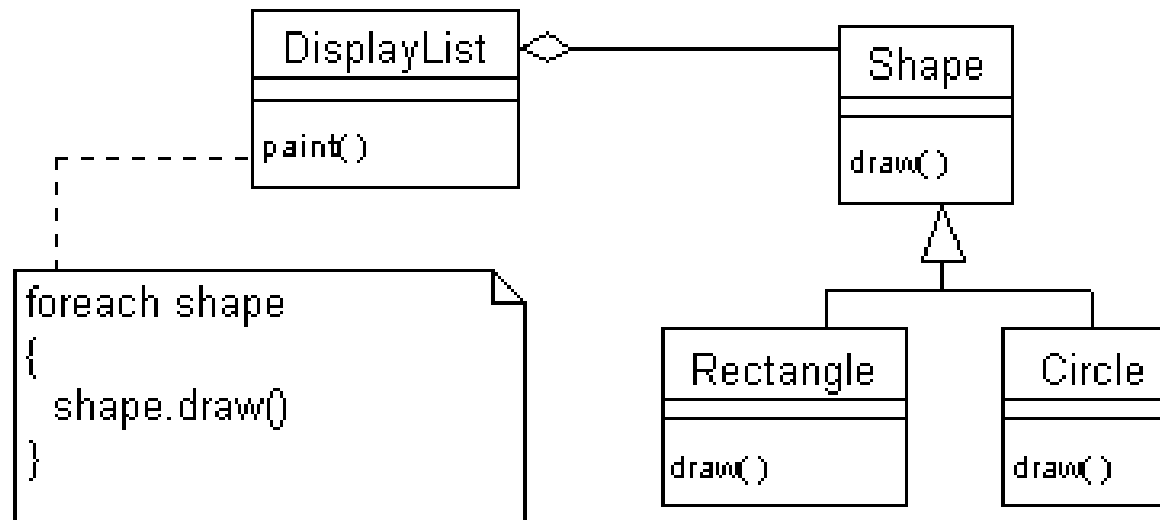
- The previous code example cannot be extended without modification:
  - Adding a switch can easily be forgotten !
  - There are dependencies from DisplayList to every sub-class of Shape and changes to working code are always a risk.
- Allowing Open for extension only:
  - New features "add new code" instead of "changing working code".
  - Leads to avoiding dynamic casting and avoids the "everyone changes the same file all the time" syndrome.
  - Error code headers are a good example ...



# Open / Closed Principle

---

– Open for extension example:



In this case we can safely add new types of shapes - these will require no change to the DisplayList class.

# Liskov Substitution Principle

---

- Subtypes are usable through the supertype interface.
  - » The calling code should not need to know the difference.
  - » If the subtype does not comply with this principle the caller may have special code.
    - Most object oriented runtime systems support this notion through polymorphism.
    - So we don't normally have to take any special measures - other than using good design !

# Dependency Inversion Principle

» Details depend on abstractions. Abstractions do not depend on details.

– Imagine a simple home heating system:

```
while (true) // Infinite loop
{
    while (temp > min)
        sleep(100);
    turnOnHeat();
    while (temp < max)
        sleep(100);
    turnOffHeater();
}
```

System is turned on if the current temp falls below min. It's then turned off when it goes above max ...

# Dependency Inversion Principle

- You can read and write to devices.
- The thermometer and heater are devices which are controlled by a Thermostat class.
- This class is instantiated and started i.e. it runs in its own thread of execution:

```
class Thermostat extends Thread
{
    private final int THERMOMETER = 0x10;
    private final int HEATER = 0xf7;
    private final int HEATER_ON = 0x1;
    private final int HEATER_OFF = 0;
```

# Dependency Inversion Principle

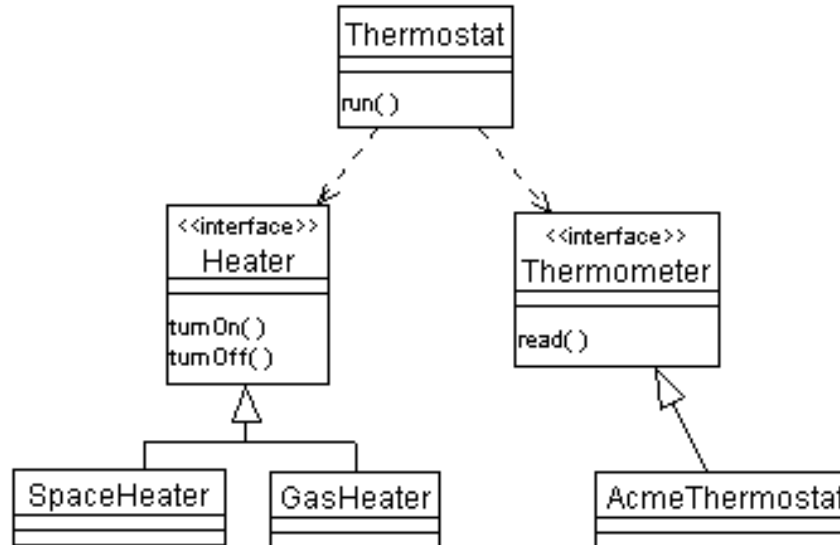
```
void run()
{
    while (read(THERMOMETER) > min)
        sleep(100);
    write(HEATER, HEATER_ON);
    while (read(THERMOMETER) < max)
        sleep(100);
    write(HEATER, HEATER_OFF);
}
```

- What's wrong with this design ? What happens when a new type of thermometer comes along ?

# Using Inversion

---

- Using inversion, the Thermostat uses two other objects Thermometer and Heater:
  - These are interfaces (abstract).
  - The thermostat is implemented using abstractions.



# Using Inversion

---

- » Consequences in this case:
  - The Thermostat is more reusable.
  - It is more extensible.
  - There is more code (cost of inversion):
    - So maybe not worth it in this small example but definitely worthwhile in a large system ...
- » As a small exercise!
  - Write the JAVA code for the inversion version.
  - Put the read / write calls in the concrete specialisations ...

# JAVA

---

- Internationalization

- » Suppose that you've written a program that displays three messages, as follows:

```
public class Not18N {  
  
    static public void main(String[] args) {  
  
        System.out.println("Hello.");  
        System.out.println("How are you?");  
        System.out.println("Goodbye.");  
    }  
}
```



# JAVA

---

- You've decided that this program needs to display these same messages for people living in France and Germany.
- Since the translators aren't programmers, you'll have to move the messages out of the source code and into text files that the translators can edit.
- Also, the program must be flexible enough so that it can display the messages in other languages, but right now no one knows what those languages will be.
- It looks like the program needs to be internationalized...

# JAVA

---

```
import java.util.*;

public class I18NSample {

    static public void main(String[] args) {

        String language;
        String country;

        if (args.length != 2) {
            language = new String("en");
            country = new String("US");
        } else {
            language = new String(args[0]);
            country = new String(args[1]);
        }
    }
}
```

# JAVA

---

```
Locale currentLocale;  
ResourceBundle messages;  
  
currentLocale = new Locale(language, country);  
  
messages = ResourceBundle.getBundle("MessagesBundle",  
                                   currentLocale);  
System.out.println(messages.getString("greetings"));  
System.out.println(messages.getString("inquiry"));  
System.out.println(messages.getString("farewell"));  
}  
}
```

# JAVA

---

- The internationalized program is flexible; it allows the end user to specify a language and a country on the command line. In the following example the language code is fr (French) and the country code is FR (France), so the program displays the messages in French:

```
% java I18NSample fr FR  
Bonjour.  
Comment allez-vous?  
Au revoir.
```

In the next example the language code is en (English) and the country code is US (United States) so the program displays the messages in English:

```
% java I18NSample en US  
Hello.  
How are you?  
Goodbye.
```

# JAVA

---

- Internationalizing the Sample Program
  - » If you look at the internationalized source code, you'll notice that the hard coded English messages have been removed.
  - » Because the messages are no longer hard coded and the language code is specified at run time, the same executable can be distributed worldwide.
  - » No recompilation is required for localization -the program has been internationalized.

# JAVA

---

- Properties Files and Locales

- » A properties file stores information about the characteristics of a program or environment and is in plain-text format.
- » In the example shown the properties files store the translatable text of the messages to be displayed.
- » The default properties file, which is called `MessagesBundle.properties`, contains the following lines:

```
greetings = Hello  
farewell = Goodbye  
inquiry = How are you?
```

# JAVA

---

- » Now that the messages are in a properties file, they can be translated into various languages.
- » The French translator has created a properties file called `MessagesBundle_fr_FR.properties`, which contains these lines:

```
greetings = Bonjour.  
farewell = Au revoir.  
inquiry = Comment allez-vous?
```

- » The name of the properties file is important. For example, the name of the `MessagesBundle_fr_FR.properties` file contains the fr language code and the FR country code.
- » These codes are also used when creating a `Locale` object.

# JAVA

---

- » The Locale object identifies a particular language and country. The following statement defines a Locale for which the language is English and the country is the United States:

```
aLocale = new Locale("en","US");
```

- » Create Locale objects for the French language:

```
caLocale = new Locale("fr","CA");  
frLocale = new Locale("fr","FR");
```

- » The sample program is flexible. Instead of using hard coded language and country codes, the program gets them from the command line at run time.



# JAVA

---

- ResourceBundle

- » Use ResourceBundle objects to isolate locale-sensitive data, such as text:

```
message = ResourceBundle.getBundle("MessagesBundle",  
    currentLocale);
```

- » The arguments passed to the getBundle method identify which properties file will be accessed:

```
MessagesBundle_en_US.properties  
MessagesBundle_fr_FR.properties  
MessagesBundle_de_DE.properties
```

- » The Locale, which is the second argument of getBundle, specifies which of the MessagesBundle files is chosen.

# JAVA

---

- Fetching Text from the ResourceBundle
  - » The properties files contain key-value pairs. The values consist of the translated text that the program will display.
  - » You specify the keys when fetching the translated messages from the ResourceBundle with the getString method:

```
String msg1 = messages.getString("greetings");
```

# JAVA

---

- java.text Package
  - » Provides classes and interfaces for handling text, dates, numbers, and messages in a manner independent of natural languages.
  - » Your application can be written to be language-independent, and it can rely upon separate, dynamically-linked localized resources.
  - » This package contains three main groups of classes and interfaces:
    - Classes for iteration over text
    - Classes for formatting and parsing
    - Classes for string collation

# JAVA

---

- Comparing Strings

- » If your application audience is limited to people who speak English, you can probably perform string comparisons with the `String.compareTo` method.
- » The `String.compareTo` method performs a binary comparison of the Unicode characters within the two strings.
  - For most languages, however, this binary comparison cannot be relied on to sort strings, because the Unicode values do not correspond to the relative order of the characters.
- » Fortunately the `Collator` class allows your application to perform string comparisons for different languages.

# JAVA

---

- Performing Locale-Independent Comparisons
  - » Collation rules define the sort sequence of strings. These rules vary with locale, because various natural languages sort words differently.
  - » You can use the predefined collation rules provided by the Collator class to sort strings in a locale-independent manner.
  - » To instantiate the Collator class invoke the getInstance method. Usually, you create a Collator for the default Locale, as in the following example:

```
Collator myDefaultCollator = Collator.getInstance();
```

# JAVA

---

- » You can also specify a particular Locale when you create a Collator, as follows:

```
Collator myFrenchCollator =  
    Collator.getInstance(Locale.FRENCH);
```

- » The getInstance method returns a RuleBasedCollator, which is a concrete subclass of Collator.
- » The RuleBasedCollator contains a set of rules that determine the sort order of strings for the locale you specify.
- » Because the rules are encapsulated within the RuleBasedCollator, your program won't need special routines to deal with the way collation rules vary with language.

# JAVA

---

- » You invoke the `Collator.compare` method to perform a locale-independent string comparison.
- » The following contains some sample calls to `Collator.compare`:
  - `myCollator.compare("abc", "def");` // returns -1 as "abc" is less than "def"
  - `myCollator.compare("rtf", "rtf");` // returns 0 as the two strings are equal
  - `myCollator.compare("xyz", "abc");` // returns 1 as "xyz" is greater than "abc"

# JAVA

---

- » The sample program called CollatorDemo uses the compare method to sort an array of English and French words.
  - This program shows what can happen when you sort the same list of words with two different collators:
  - `Collator fr_FRCollator = Collator.getInstance(new Locale("fr","FR"));`
  - `Collator en_USCollator = Collator.getInstance(new Locale("en","US"));`
- » The method for sorting, called `sortStrings`, can be used with any Collator. Notice that the `sortStrings` method invokes the compare method:



# JAVA

---

```
public static void sortStrings(Collator collator,  
                               String[] words) {  
    String tmp;  
    for (int i = 0; i < words.length; i++) {  
        for (int j = i + 1; j < words.length; j++) {  
            if (collator.compare(words[i], words[j]) > 0) {  
                tmp = words[i];  
                words[i] = words[j];  
                words[j] = tmp;  
            }  
        }  
    }  
}
```

# JAVA

---

- » The English Collator sorts the words as follows:
  - peach
  - péché
  - pêche
  - sin
- » According to the collation rules of the French language, the preceding list is in the wrong order. In French péché should follow pêche in a sorted list.
- » The French Collator sorts the array of words correctly, as follows:
  - peach
  - pêche
  - péché
  - sin