Date	Chapter
11/6/2006	Chapter 10, start **Chapter 11**
11/13/2006	Chapter 11, start Chapter 12
11/20/2006	Chapter 12
11/27/2006	Chapter 13
12/4/2006	Final Exam
12/11/2006	Project Due

Chapter 11

Exceptions
and
Input/Output Operations

Topics

- Exception Handling
  - Using ***try*** and ***catch*** Blocks
  - Catching Multiple Exceptions
  - User-Defined Exceptions
- The *java.io* Package
- Reading from the Java Console
- Reading and Writing Text Files
- Reading Structured Text Files Using ***StringTokenizer***
- Reading and Writing Objects to a File

Exceptions

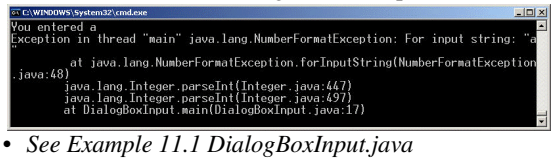
- Java is robust language and does not allow Illegal operations at run time to occur, they generate exceptions, for example:
  - ***ArrayIndexOutOfBoundsException***
  - ***ArithmeticException***
  - ***NullPointerException***
  - ***InputMismatchException***
  - ***NumberFormatException***

Handling Exceptions

- In a program without a Graphical User Interface, exceptions cause the program to terminate.
- With this code:

```
12 String s = JOptionPane.showInputDialog( null,
13                 "Enter an integer" );
…
17 int n = Integer.parseInt( s );
```

- If the user enters "a", we get this exception:



- *See Example 11.1 DialogBoxInput.java*

Handling Exceptions

- We don't want invalid user input to terminate the program!
- It is better to detect the problem and reprompt the user for the input.
- We can intercept and **handle** some of these exceptions using *try* and *catch* blocks.
  - Inside the ***try*** block, we put the code that might generate an exception.
  - Inside ***catch*** blocks, we put the code to handle any exceptions that could be generated.
- Java provides exception classes and try, catch, and finally blocks to support exceptions

Minimum *try/catch* Syntax

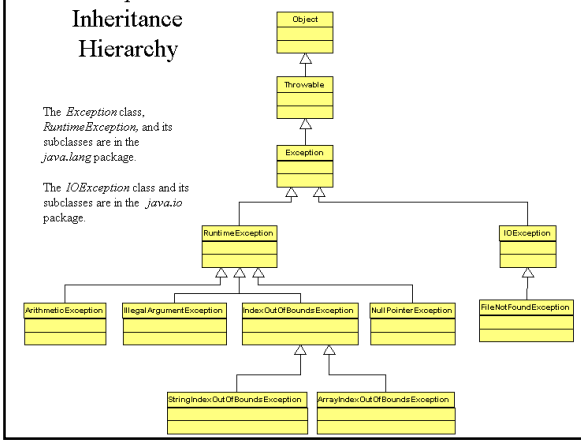
```
try
{
   // code that might generate an exception
}
catch( ExceptionClass exceptionObjRef )
{
   // code to recover from the exception
}
```

- If an exception occurs in the *try* block, the *try* block terminates and control jumps immediately to the *catch* block.
- If no exceptions are generated in the *try* block, the *catch* block is not executed.

Exception Inheritance Hierarchy

The *Exception* class, *RuntimeException,* and its subclasses are in the *java.lang* package.

The *IOException* class and its subclasses are in the *java.io* package.



Checked and Unchecked Exceptions

- Java distinguishes between two types of exceptions:
- **Unchecked exceptions** are those that are subclasses of *Error* or *RuntimeException*
  - It is not mandatory to use try and catch blocks to handle these exceptions.
  - If you omit try and catch blocks your code will compile.
  - If they occur the JVM will catch and display the error message.
  - ***ArithmeticException*** (caused by divide by zero), ***NumberFormatException, NullPointerException***
- **Checked exceptions** are any other exceptions.
  - Code that might generate a checked exception must be put inside a *try* block. Otherwise, the compiler will generate an error.
  - ***IOException***

Exception Class

- Is the super class of all exception classes, it contains many predefined exceptions such as:
  - Integer divide by zero
  - Out-of-bound array index
  - Illegal number format
  - File does not exist
  - Etc.

Exception Class Methods

- Inside the ***catch*** block, you can call any of these methods of the *Exception* class:

Return value	Method name and argument list
`String`	`getMessage( )` returns a message indicating the cause of the exception
`String`	`toString( )` returns a String containing the exception class name and a message indicating the cause of the exception
`void`	`printStackTrace( )` prints the line number of the code that caused the exception along with the sequence of method calls leading up to the exception

Catching a *NumberFormatException*

```
int n = 0;  // declare and initialize variable
String s = JOptionPane.showInputDialog( null,
                    "Enter an integer" );
try
{
   n = Integer.parseInt( s );
   System.out.println( "You entered " + n );
}
catch ( NumberFormatException nfe )
{
  System.out.println( "Incompatible data." );
}
```

- *See Example 10.2 DialogBoxInput.java*

public static int parseInt (String str) throws NumberFormatExeption

Initializing Variables for *try/catch* Blocks

- Notice that we declare and initialize the input variable before we enter the *try* block. If we do not initialize the variable and then try to access it after the *try/catch* blocks, we will receive the following compiler error:

  `variable n might not have been initialized`

  The error indicates that the only place where *n* is assigned a value is in the *try* block. If an exception occurs, the *try* block will be interrupted and we might not ever assign *n* a value.
- Initializing the value before entering the *try* block solves this problem.

Recovering From an Exception

- The previous code just printed a message when the exception occurred.
- To continue processing and reprompt the user for good input, we can put the *try* and *catch* blocks inside a *do/while* loop.
- *See Example 11.3 DialogBoxInput.java (next Slide)*

```
int n = 0;
boolean goodInput = false; // flag variable

String s = JOptionPane.showInputDialog( null,
                        "Enter an integer" );
do {
  try {
      n = Integer.parseInt( s );
      goodInput = true; //executed if no exception
  }
  catch ( NumberFormatException nfe ) {
      s = JOptionPane.showInputDialog( null,
                      s + " is not an integer. "
                      + "Enter an integer" );
  }
} while ( ! goodInput );
```

Software Engineering Tip

Write code to catch and handle exceptions generated by invalid user input.

Although the methods of the *Exception* class are good debugging tools, they are not necessarily appropriate to use in the final version of a program.

Always try to write code that is user-friendly.

Catching Multiple Exceptions

- If the code in the *try* block might generate multiple, different exceptions, we can provide multiple *catch* blocks, one for each possible exception.
- When an exception is generated, the JVM searches the *catch* blocks in order. The **first *catch*** block with a parameter that matches the exception thrown will execute; any remaining *catch* blocks will be skipped.

catch Block Order

- An exception will match any *catch* block with a parameter that names any of its superclasses.
  - For example, a *NumberFormatException* will match a *catch* block with a *RuntimeException* parameter.
  - All exceptions will match a *catch* block with an *Exception* parameter.
- Thus, when coding several *catch* blocks, arrange the *catch* blocks with the specialized exceptions first, followed by more general exceptions.

The *finally* Block

- Optionally, you can follow the *catch* blocks with a ***finally*** block.
- The *finally* block will be executed whether or not an exception occurs. Thus:
  - if an exception occurs, the *finally* block will be executed when the appropriate *catch* block finishes executing
  - if no exception occurs, the *finally* block will be executed when the *try* block finishes
- For example, a *finally* block might be used to close an open file. We demonstrate this later.

Full *try/catch/finally* Syntax

```
try
{
   // code that might generate an exception
}
catch( Exception1Class e1 )
{
   // code to handle an Exception1Class exception
}
…
catch( ExceptionNClass eN )
{
   // code to handle an ExceptionNClass exception
}
finally
{
  // code to execute in any case
}
```

Catching Multiple Exceptions

- We can write a program that catches several exceptions.
- For example, we can prompt the user for a divisor.
  - If the input is not an integer, we catch the *NumberFormatException* and reprompt the user with an appropriate message.
  - If the input is 0, we catch an *ArithmeticException* when we attempt to divide by 0, and reprompt the user with an appropriate message.
- *See Example 11.4 Divider.java*

User-Defined Exceptions

- We can design our own exception class.
- Suppose we want to design a class encapsulating email addresses (*EmailAddress* class).
  - For simplicity, we say that a legal email address is a *String* containing the @ character.
- Our *EmailAddress* constructor will throw an exception if its email address argument is illegal.
- To do this, we design an exception class named *IllegalEmailException*.

User-Defined Exception

- Java has an *IllegalArgumentException* class, so our *IllegalEmailException* class can be a subclass of the *IllegalArgumentException* class.
- By extending the *IllegalArgumentException* class:
  - we inherit the functionality of an exception class, which simplifies our coding of the exception
  - we can associate a specific error message with the exception

Extending an Existing Exception

- We need to code only the constructor, which accepts the error message as a *String*.
- General pattern:

```
public class ExceptionName
        extends ExistingExceptionClassName
{
  public ExceptionName( String message )
  {
    super( message );
  }
}
```

- *See Example 11.5 IllegalEmailException.java*

Throwing an Exception

- The pattern for a method that throws a user-defined exception is:

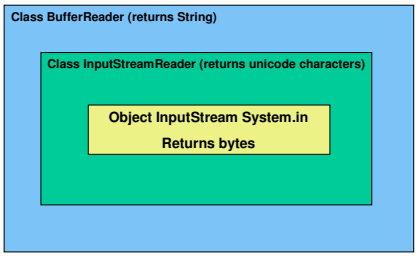
```
accessModifier returnType methodName( parameters )
                    throws ExceptionName
{
  if( parameter list is legal )
    process the parameter list
  else
    throw new ExceptionName( "Message here" );
}
```

- The message passed to the constructor identifies the error we detected. In a client's *catch* block, the *getMessage* method will retrieve that message.
- *See Examples 11.6 & 11.7*

java.io Package

A set of classes used for:
reading and writing from files
reading from console

System.in Object

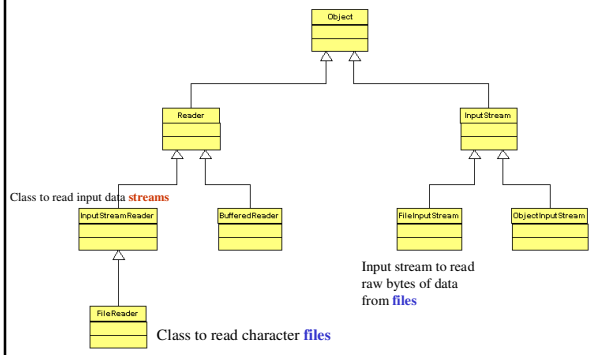



```
BufferedReader inStream = new BufferedReader(new InputStreamReader(System.in));
```

Selected Input Classes in the *java.io* Package

Class	Description
`Reader`	*Abstract* superclass for input classes
`InputStream`	*Abstract* superclass representing a stream of **raw bytes**
`InputStreamReader`	Class to read input data **streams**
`FileReader`	Class to read character **files**
`BufferedReader`	Class providing more efficient reading of character files
`FileInputStream`	Input stream to read raw bytes of data from **files**
`ObjectInputStream`	Class to read/recover objects from a file written using *ObjectOutputStream*

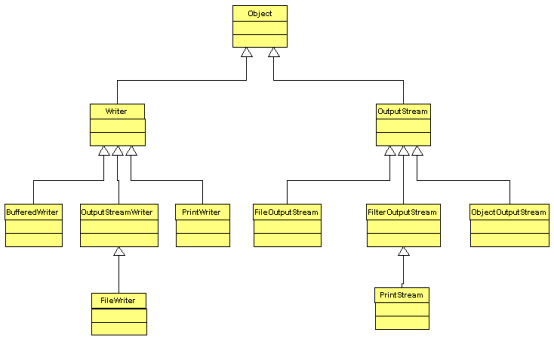
Hierarchy for Input Classes



Selected *java.io* Output Classes

Class	Description
`Writer`	*Abstract* superclass for output classes
`OutputStreamWriter`	Class to write output data streams
`OutputStream`	*Abstract* superclass representing an output stream of raw bytes
`FileWriter`	Class for writing to character files
`BufferedWriter`	More efficient writing to character files
`PrintWriter`	Prints basic data types, *Strings*, and objects
`PrintStream`	Supports printing various data types conveniently
`FileOutputStream`	Output stream for writing raw bytes of data to files
`ObjectOutputStream`	Class to write objects to a file

Hierarchy for Output Classes



Reading from the Java Console

- *System.in* is the default standard input device, which is tied to the Java Console.
- We have read from the console by associating a *Scanner* object with the standard input device:

```
Scanner scan = new Scanner( System.in );
```

- We can also read from the console using these subclasses of *Reader:*
  - *InputStreamReader*
  - *BufferedReader,* uses buffering (read-ahead) for efficient reading

Opening an InputStream

- When we construct an input stream or output stream object, the JVM associates the file name, standard input stream, or standard output stream with our object. This is **opening the file.**
- When we are finished with a file, we optionally call the *close* method to release the resources associated with the file.
- In contrast, the standard input stream (*System.in*), the standard output stream (*System.out*), and the standard error stream *(System.err*) are open when the program begins. They are intended to stay open and should not be closed.

Software Engineering Tip

Calling the close method is optional. When the program finishes executing, all the resources of any unclosed files are released.

It is good practice to call the *close* method, especially if you will be opening a number of files (or opening the same file multiple times.)

Do not close the standard input, output, or error devices, however. They are intended to remain open.

Console Input Class Constructors

Class	Constructor
`InputStreamReader`	`InputStreamReader( InputStream is )` constructs an *InputStreamReader* object from an *InputStream* object. For console input, the *InputStream* object is *System.in.*
`BufferedReader`	`BufferedReader( Reader r )` constructs a *BufferedReader* object from a *Reader* object – here the *Reader* object will be an *InputStreamReader* object.

Methods of the *BufferedReader* Class

Return value	Method name and argument list
`String`	`readLine( )` reads a line of text from the current *InputStream* object, and returns the text as a *String*. Throws an *IOException*.
`void`	`close( )` releases resources associated with an open input stream. Throws an *IOException*.

- Because an *IOException* is a checked exception, we must call these methods within a *try* block.
- ***See Example 11.8 ConsoleInput.java***

Alternative Coding

- This code:

```
InputStreamReader isr =
        new InputStreamReader( System.in );
BufferedReader br = new BufferedReader( isr );
```

can also be coded as one statement using an anonymous object:

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

because the object reference *isr* is used only once.

Hiding the Complexity

- We can hide the complexity by encapsulating *try* and *catch* blocks into a *UserInput* class, which is similar in concept to the *Scanner* class.
- We write our class so that the client program can retrieve user input with just one line of code.
- The *UserInput* class also validates that the user enters only the appropriate data type and reprompts the user if invalid data is entered.
- *See Examples 11.9 and 11.10*

Software Engineering Tip

Encapsulate complex code into a reusable class. This will simplify your applications and make the logic clearer.

File Types

- Java supports two types of files:
  - text files: data is stored as characters
  - binary files: data is stored as raw bytes
- The type of a file is determined by the classes used to write to the file.
- To read an existing file, you must know the **file's type** in order to select the appropriate classes for reading the file.

Reading Text Files

- A text file is treated as a stream of characters.
- ***FileReader*** is designed to read character files.
- A *FileReader* object does not use buffering, so we will use the ***BufferedReader*** class and the *readLine* method to read more efficiently from a text file.

Constructors for Reading Text Files

Class	Constructor
`FileReader`	`FileReader( String filename )` constructs a *FileReader* object from a *String* representing the name of a file. Throws a *FileNotFoundException*.
`BufferedReader`	`BufferedReader( Reader r )` constructs a *BufferedReader* object from a *Reader* object

Methods of the *BufferedReader* Class

Return value	Method name and argument list
`String`	`readLine( )` reads a line of text from the current *InputStream* object, and returns the text as a *String*. Returns a *null String* when the end of the file is reached. Throws an *IOException*.
`void`	`close( )` releases resources allocated to the *BufferedReader* object. Throws an *IOException*.

- *See Example 11.11 ReadTextFile.java*

Writing to Text Files

- Several situations can exist:
  - the file does not exist
  - the file exists and we want to replace the current contents
  - the file exists and we want to append to the current contents
- We specify whether we want to replace the contents or append to the current contents when we construct our *FileWriter* object.

Constructors for Writing Text Files

Class	Constructor
`FileWriter`	`FileWriter( String filename, boolean mode )` constructs a *FileWriter* object from a *String* representing the name of a file. If the file does not exist, it is created. If *mode* is *false*, the current contents of the file, if any, will be replaced. If *mode* is *true*, writing will append data to the end of the file. Throws an *IOException*.
`BufferedWriter`	`BufferedWriter( Writer w )` constructs a *BufferedWriter* object from a *Writer* object

Methods of the *BufferedWriter* Class

Return value	Method name and argument list
`void`	`write( String s )` writes a *String* to the current *OutputStream* object. This method is inherited from the *Writer* class. Throws an *IOException*.
`void`	`newLine( )` writes a line separator. Throws an *IOException*.
`void`	`close( )` releases resources allocated to the *BufferedWriter* object. Throws an *IOException*.

- *See Examples 11.12 & 11.13*

Reading Structured Text Files

- Some text files are organized into lines that represent a **record --** a set of data values containing information about an item.
- The data values are separated by one or more **delimiters;** that is, a special character or characters separate one value from the next.
- As we read the file, we need to **parse** each line; that is, separate the line into the individual data values called **tokens**.

Example

- An airline company could store data in a file where each line represents a flight segment containing the following data:
  - flight number
  - origin airport
  - destination airport
  - number of passengers
  - average ticket price
- Such a file could contain the following data:

```
AA123,BWI,SFO,235,239.5
AA200,BOS,JFK,150,89.3
AA900,LAX,CHI,201,201.8
…
```

- In this case, the delimiter is a comma.

The *StringTokenizer* Class

- The ***StringTokenizer*** class is designed to parse *Strings* into tokens.
- ***StringTokenizer*** is in the *java.util* package.
- When we construct a *StringTokenizer* object, we specify the delimiters that separate the data we want to tokenize. The default delimiters are the whitespace characters.

Two *StringTokenizer* Constructors

Constructor name and argument list
`StringTokenizer( String str )` constructs a *StringTokenizer* object for the specified *String* using space, tab, carriage return, newline, and form feed as the default delimiters
`StringTokenizer( String str, String delim )` constructs a *StringTokenizer* object for the specified *String* using *delim* as the delimiters

Useful *StringTokenizer* Methods

Return value	Method name and argument list
`int`	`countTokens( )` returns the number of unretrieved tokens in this object; the count is decremented as tokens are retrieved.
`String`	`nextToken( )` returns the next token
`boolean`	`hasMoreTokens( )` returns *true* if more tokens are available to be retrieved; returns *false*, otherwise.

Using *StringTokenizer*

```
import java.util.StringTokenizer;
public class UsingStringTokenizer
{
  public static void main( String [] args )
  {
   String flightRecord1 = "AA123,BWI,SFO,235,239.5";
   StringTokenizer stfr1 =
        new StringTokenizer( flightRecord1, "," );
        // the delimiter is a comma

   while ( stfr1.hasMoreTokens( ) )
     System.out.println( stfr1.nextToken( ) );
  }
}
```

- *See Example 11.14 UsingStringTokenizer.java*

Common Error Trap

Why didn't we use a *for* loop and the *countTokens* method?

```
for ( int i = 0; i < strfr1.countTokens( ); i++ )
    System.out.println( stfr1.nextToken( ) );
```

This code won't work because the return value of *countTokens* is the number of tokens **remaining to be retrieved.**

The body of the loop retrieves one token, so each time we evaluate the loop condition by calling the *countTokens* method, the return value is 1 fewer.

The result is that we retrieve only half of the tokens.

Example Using *StringTokenizer*

- The file *flight.txt* contains the following comma-separated flight data on each line:

```
flight number, origin airport, destination airport,
  number of passengers, average ticket price
```

- The *FlightRecord* class defines instance variables for each flight data value
- The *ReadFlights* class reads data from *flights.txt,* instantiates *FlightRecord* objects, and adds them to an *ArrayList*.
- *See Examples 11.15 & 11.16*

Writing Primitive Types to Text Files

- *FileOutputStream*, a subclass of the *OutputStream* class, is designed to write a stream of bytes to a file.
- The *PrintWriter* class is designed for converting primitive data types to characters and writing them to a text file.
  - *print* method, writes data to the file without a newline
  - *println* method, writes data to the file, then adds a newline

Constructors for Writing Structured Text Files

Class	Constructor
`FileOutputStream`	`FileOutputStream( String filename, boolean mode )` constructs a *FileOutputStream* object from a *String* representing the name of a file. If the file does not exist, it is created. If *mode* is *false*, the current contents of the file, if any, will be replaced. If *mode* is *true*, writing will append data to the end of the file. Throws a *FileNotFoundException*.
`PrintWriter`	`PrintWriter( OutputStream os )` constructs a *PrintWriter* object from an *OutputStream* object

Useful *PrintWriter* Methods

Return value	Method name and argument list
`void`	`print( dataType argument )` writes a *String* representation of the argument to the file.
`void`	`println( dataType argument )` writes a *String* representation of the argument to the file followed by a newline.
`void`	`close( )` releases the resources associated with the *PrintWriter* object

- The argument can be any primitive data type (except *byte* or *short*), a *char* array, or an object.
- *See Example 11.18 WriteGradeFile.java*

Reading and Writing Objects

- Java also supports writing objects to a file and reading them as objects.
- This is convenient for two reasons:
  - We can write these objects directly to a file without having to convert the objects to primitive data types or *Strings*.
  - We can read the objects directly from a file, without having to read *Strings* and convert these *Strings* to primitive data types in order to instantiate objects.
- To read objects from a file, the objects must have been written to that file as objects.

Writing Objects to a File

- To write an object to a file, its class must implement the ***Serializable*** interface, which indicates that:
  - the object can be converted to a byte stream to be written to a file
  - that byte stream can be converted back into a copy of the object when read from the file.
- The *Serializable* interface has no methods to implement. All we need to do is:
  - *import* the *java.io.Serializable* interface
  - add *implements Serializable* to the class header

The *ObjectOutputStream* Class

- The ***ObjectOutputStream*** class, coupled with the ***FileOutputStream*** class, provides the functionality to write objects to a file.
- The ***ObjectOutputStream*** class provides a convenient way to write objects to a file.
  - Its *writeObject* method takes one argument: the object to be written.

Constructors for Writing Objects

Class	Constructor
`FileOutputStream`	`FileOutputStream( String filename, boolean mode )` creates a *FileOutputStream* object from a *String* representing the name of a file. If the file does not exist, it is created. If *mode* is *false*, the current contents of the file, if any, will be replaced. If *mode* is *true*, writing will append data to the end of the file. Throws a *FileNotFoundException*.
`ObjectOutputStream`	`ObjectOutputStream( OutputStream out )` creates an *ObjectOutputStream* that writes to the *OutputStream out*. Throws an *IOException*.

The *writeObject* Method

Return value	Method name and argument list
`void`	`writeObject( Object o )` writes the object argument to a file. That object must be an instance of a class that implements the *Serializable* interface. Otherwise, a run-time exception will be generated. Throws an *IOException*.

- *See Examples 11.19 & 11.20*

Omitting Data from the File

- The *writeObject* method does not write any object fields declared to be *static* or ***transient.***
- You can declare a field as *transient* if you can easily reproduce its value or if its value is 0.
  - Syntax to declare a field as *transient*:

```
accessModifier transient dataType fieldName
```

  - Example:

```
private transient double totalRevenue;
```

Software Engineering Tip

To save disk space when writing to an object file, declare the class's fields as *static* or *transient,* where appropriate.

Reading Objects from a File

- The *ObjectInputStream* class, coupled with *FileInputStream*, provides the functionality to read objects from a file.
- The *readObject* method of the *ObjectInputStream* class is designed to read objects from a file.
- Because the *readObject* method returns a generic *Object*, we must type cast the returned object to the appropriate class.
- When the end of the file is reached, the *readObject* method throws an *EOFException*, so we detect the end of the file when we catch that exception.

Constructors for Reading Objects

Class	Constructor
`FileInputStream`	`FileInputStream( String filename )` constructs a *FileInputStream* object from a *String* representing the name of a file. Throws a *FileNotFoundException*.
`ObjectInputStream`	`ObjectInputStream( InputStream in )` creates an *ObjectInputStream* from the *InputStream in*. Throws an *IOException*.

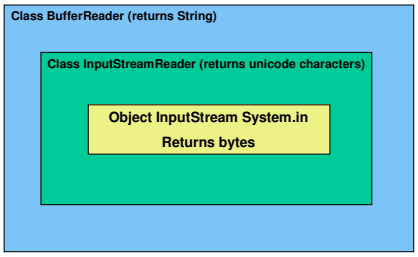
The *readObject* Method

Return value	Method name and argument list
`Object`	`readObject( )` reads the next object and returns it. The object must be an instance of a class that implements the *Serializable* interface. When the end of the file is reached, an *EOFException* is thrown. Also throws an *IOException* and *ClassNotFoundException*

- *See Example 11.21 ReadingObjects.java*
  - Note that we use a *finally* block to close the file.

Backup

System.in Object




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

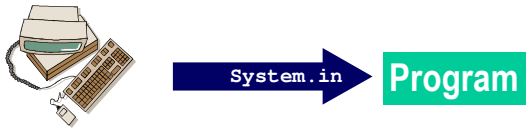
Input Streams

- Stream is flow of data
  - Reader at one end
  - Writer at the other end

- Stream generalizes input & output
  - Keyboard electronics different from disk
  - Input stream makes keyboard look like a disk

Input Streams: System.in

- System.in: the standard input stream
  - By default, reads characters from the keyboard



- Can use System.in many ways
  - Directly (low-level access)
  - Through layers of abstraction (high-level access)

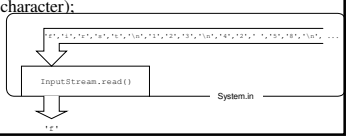
Input Streams: Read Characters

- Can read characters from System.in with read()

```
// Reads a single character from the keyboard and displays it
class DemonstrateRead
{
  public static void main(String[] args)
  throws java.io.IOException
  {
    char character;

    // Prompt for a character and read it
    System.out.print("Enter a character: ");
    System.out.flush();
    character = (char) System.in.read();

    // Display the character typed
    System.out.println();
    System.out.println("You typed " + character);
  }
}
```



Input Streams: Read Numbers

- Can combine reading, parsing, conversion steps

```
import java.io.*;
import java.text.NumberFormat;
class ReadAnInt2 {
  public static void main(String[] args)
  throws java.io.IOException, java.text.ParseException  {
// Create an input stream and attach it to the standard input stream
    BufferedReader inStream
      = new BufferedReader(new InputStreamReader(System.in));
    // Create a number formatter object
    NumberFormat aNumberFormatter = NumberFormat.getInstance();
    System.out.print("Enter an integer: ");
// Read the response from the user, convert to Number,then convert to int
    int intNumber
      = aNumberFormatter.parse(inStream.readLine()).intValue();
System.out.println("You typed " + intNumber);
  }
}
```

Note ParseException!