Wrapper Classes

❖ Instantiable classes representing the primitive data types, including:

- Byte
- Short
- Integer
- Long
- Float
- Double
- Boolean

❖ Important methods of the Wrapper Classes:

```
public int intValue()
// returns an int, the value of the Integer object

public boolean equals()
// compares two Integer objects

public static int parseInt()
// converts a String object into an int
```

❖ All Wrapper classes are immutable like the String class

- once created, cannot be changed
- rather than being changed, a whole new object is created reflecting that change

❖ Examples:

```
Integer integer1 = new Integer(1);
Double double2 = new Double(2.0);
```

❖ Usefulness of Wrapper classes:

- Whenever only objects are taken, need a way to represent the primitive data types as objects
- Example: the Vector class

```
Vector v = new Vector (10);
for (int i = 0; i < v.capacity(); i++) {
    v.addElement (new Integer (i));
}
```

Java Input/Output

❖ In General, "streams" are used.

- A stream is a sequence of data (8-bit bytes)
  - Input Stream: brings information to a program from some source
  - Output Stream: sends information from a program to some destination

Input Stream Sources	Output Stream Destinations
Standard Input	Standard Output
Files	Standard Error
etc...	Files
	etc...

File input: "read" data from a file
File output: "save" data to a file

- The package java.io provides two class hierarchies that support streams
  - one is for byte streams (Standard input, output, error)
  - one is for character streams (handles characters > 1 byte in size)

  NOTE: must include the statement: **`import java.io.*;`**

- The particular class used depends on the following:
  - read vs. write
  - human readable (e.g., ascii) vs. binary (e.g., 1001100011)

❖ Console I/O (i.e., standard input/output)

- Console Output
  - **`System.out.print(<arg>) & System.out.println(<arg>)`**
  - Example:

```
System.out.print("Hello");
System.out.println(" World!");
```

- Console Input
  - **`System.in`** takes in raw data (in bytes) from the standard input (keyboard)
  - Create **`InputStreamReader`** and **`BufferedReader`** objects to convert the raw data to a readable form

MEMORIZE THIS LINE OF CODE

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

  - methods of the BufferedReader class:

```
// to read one line of text (up to newline)
// returns a String object
stdin.readLine();

// do not use; rather, only realize that it exists
// (reads in one character but returns that
//  character as an int)
stdin.read();
```

  - To input a char, use the following:

```
char c = (stdin.readLine()).charAt(0);
```

  - To input other primitive types, use the Wrapper classes' static methods:

```
int i = Integer.parseInt (stdin.readLine());
long l = Long.parseLong (stdin.readLine());
float f = Float.parseFloat (stdin.readLine());
double d = Double.parseDouble (stdin.readLine());
```

  - To input a boolean (there is no parseBoolean method):

```
boolean b = (new Boolean
             (stdin.readLine())).boolValue();
```

  - Example:

```
System.out.print ("Enter a number => ");
int i = Integer.parseInt (stdin.readLine());
System.out.println ("You entered " + i);
```

❖ File I/O

- Requires a file. (can be a directory or a "regular" file)
- The File class (also in java.io.* package) is used to create a File object
- Examples of creating File objects:

```
// default is to look in current directory
File srcFile = new File ("inputFile");
File dstFile = new File ("outputFile");

// to specify a specific directory, use full path name
File srcFile = new File ("U:\\private\\inputFile");
File dstFile = new File ("U:\\private\\outputFile");
```

NOTES:

- if file does not exist, the File object is still created, but its value is set to **`null`**.
- One can also obtain the file to be associated with the File object from user input—either standard input OR using a FileDialog object

- methods of the File class:

```
boolean canRead()     /* true iff the file can be
                         read */

boolean canWrite()    /* true iff the file can be
                         written */

boolean exists()      /* true iff the file already
                         exists */

boolean isDirectory() /* true iff the file is a
                           directory */

boolean isFile()      /* true iff the file is not a
                        directory */
```

- File output:

```
PrintWriter out = new PrintWriter (new
BufferedWriter (new FileWriter (dstFile)));
```

Classes involved (an object of each class must be created):

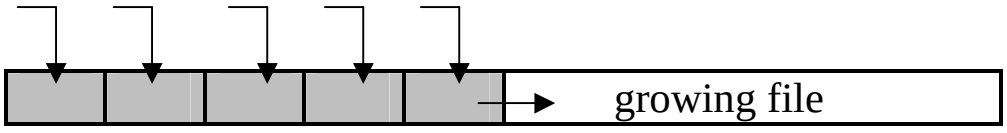
- **`FileWriter`**: outputs bytes
- **`BufferedWriter`**: places output into a buffer in blocks so that the file is not accessed for each output request (called a buffer cache)
- **`PrintWriter`**: puts output into human readable form; contains usable methods:

  `print()`: overloaded for all primitives & objects (via toString())
  `println()`: same as print, but with **`\n`** at end

  `flush()`: if output is buffered, flush the buffer
  `close()`: flush the buffer AND close the output stream
  `checkError()`: true iff an error has occurred during output

NOTES:

- Creation of a "Writer" object will automatically open the file associated with it.
- One must close a file using the **`close()`** method after writing to it in order to ensure that the data written is saved to the file.
- Files are written IN-ORDER (i.e., sequential). The next location for writing to the file is updated automatically



- File input:

```
BufferedReader in = new BufferedReader (new
FileReader (srcFile)));
```

Classes involved (an object of each class must be created):

- **`FileReader`**: reads bytes from the given File

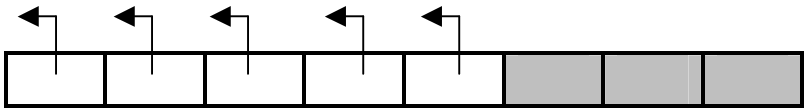
- **`BufferedReader`**: places input into a buffer in blocks so that the file is not accessed for each input request (called a buffer cache); contains usable methods:

  `readLine()`: returns a **`String`** object
  `read()`: returns an **`int`** representation of a character

  `ready()`: true iff the file has more to read
  `close()`: close the input stream

NOTES:

- Creation of a "Reader" object will automatically open the file associated with it.
- One must close a file using the **`close()`** method after reading from it.
- Files are read IN-ORDER (i.e., sequential). The next location for reading from the file is updated automatically



** The use of buffers for input and output results in more efficient file access

Summary of I/O objects and the arguments used in their constructors

CLASS	CONSTRUCTOR ARG TYPES
BufferedWriter	Writer
FileWriter	File or String
PrintWriter	OutputStream or Writer
BufferedReader	Reader
FileReader	File or String
File	String

```
/*************************************************************
Author:           Jim Skrentny, skrentny@cs.wisc.edu
                  copyright 1999, all rights reserved
Version:          1.1
Compiler:         CodeWarrior (JDK 1.2)
Platform:         Windows 95
*************************************************************/

// This import is required when doing console input.
import java.io.*;

/**
 * This program demonstrates how to do console input and output in
 * Java.  Java console I/O can be complicated to understand, but you
 * can get started by following and copying the code examples below.
 *
 * BUGS: none known
 **/

public class ConsoleIO {

    // The "throws IOException" clause is required in methods having
    // java input or any methods that directly or indirectly call
    // others having console input.  Later in class you'll learn
    // that this isn't required if the method catches IOExceptions.
    public static void main ( String args[]) throws IOException {

        // Java console output is done using System.out, a
        // PrintStream object, that is predefined in the java.lang
        // package.  Two common methods are println and print, which
        // take a string argument and display it in the console
        // window.  println causes output to continue on the next
        // line whereas print doesn't.
        System.out.print("This example " + "program ");
        System.out.println("demonstrates:");
        System.out.println("CONSOLE INPUT & OUTPUT\n\n");

        // Data is input using a BufferedReader object that is
        // composed from several different objects.  The source of
        // console input is System.in, an InputStream object, that
        // is predefined in the java.lang package.  This source must
        // have a mechanism for accessing the data.  Thus an
        // InputStreamReader object is constructed from System.in.
        // From this InputStreamReader object a BufferedReader is
        // constructed to more efficiently access the data.  Here is
        // the code:
        BufferedReader stdin = new BufferedReader( new
                               InputStreamReader ( System.in));
```

```
// With our object stdin, we can read a line of input as a
// String using the readLine method:
System.out.print("Enter a string: ");
String s = stdin.readLine();
System.out.println("You've entered: " + s);

// If you only wanted the first character of the input, use:
char c = s.charAt(0);
System.out.println("The first character is: " + c);

System.out.println("\n\n");

// If you want to read a number the input string needs to be
// converted into the desired numeric type.  This is done by
// using wrapper classes associated with each primitive
// type.  These wrapper classes have parse methods that do
// the conversions.

// Reading integers requires the input string to be
// converted to a value that is assigment compatible with
// the primitive variable.
System.out.print("Enter an integer of 3 digits: ");
int i = Integer.parseInt(stdin.readLine());
System.out.println("You've entered: " + i);
System.out.print("Enter a long integer of 11 digits: ");
long l = Long.parseLong(stdin.readLine());
System.out.println("You've entered: " + l);

System.out.println("\n\n");

// Reading real numbers is a bit more complicated and
// depends on the version of Java you have.  Most will have
// an older Java version that requires the approach below.
// In this approach a temporary Float or Double object is
// created using the wrapper classes.  This object is asked
// for its value, which is then assigned to the primitive
// variable.
System.out.print("Enter a float: ");
float f =  (new Float(stdin.readLine())).floatValue();
System.out.println("You've entered: " + f);
System.out.print("Enter a double: ");
double d = new Double(stdin.readLine()).doubleValue();
System.out.println("You've entered: " + d);
```

```
        // The code below follows the parse method approach that was
        // used with integers.  If you have the newest version of
        // Java, this code will work after you uncomment it.
        /*
        System.out.print("Enter a float: ");
        float f = Float.parseFloat(stdin.readLine());
        System.out.println("You've entered: " + f);
        System.out.print("Enter a double: ");
        double d = Double.parseDouble(stdin.readLine());
        System.out.println("You've entered: " + d);
        */

        // This is how you can read a string from the console and
        // convert it into the desired type of data.  By copying the
        // code examples above, you can now use console input and
        // output in your programs.
    }

}
```

```
/*************************************************************
Author:           Jim Skrentny, skrentny@cs.wisc.edu
                  copyright 1999, all rights reserved
Version:          1.2
Compiler:         CodeWarrior (JDK 1.2)
Platform:         Windows 95
*************************************************************/

import java.io.*;                       // for java file I/O
import java.util.StringTokenizer;       // to divide line into pieces

/**
 * This program demonstrates how to do file input and output in Java
 * using character streams and text files.  Java file I/O can be
 * complicated to understand, but you can get started by following
 * and copying the code examples below.
 *
 * BUGS: none known
 **/
public class FileIO {

    // The "throws IOException" clause is required in methods having
    // java file I/O or any methods that directly or indirectly call
    // others having file I/O.  Later in class you'll learn that
    // this isn't required if the method catches IOExceptions.
    public static void main (String args[]) throws IOException {

        // This program uses console I/O.  Please see the
        // ConsoleIO.java example program before looking at this
        // one.

        System.out.println("Example program demonstrating:" +
                           "\nFILE IO\n");

        System.out.println("BEFORE PROCEEDING: create a file" +
                           " with these 2 lines:");

        System.out.println("Dolphin|1999|");
        System.out.println("Lion|2000|");

        System.out.println("\nUse this file's path for the" +
                           " source file path.");

        // Create a BufferedReader object for console input.
        BufferedReader stdin = new BufferedReader(
                               new InputStreamReader (System.in));
```

```
// Java file input and output begins with the actual file
// stored in the computer's file system.  These files are
// the source or destination of the characters that are
// process by a program.  We'll access files by first
// creating a File object that is associated with the path
// where the actual file is located in the computer's file
// system.

// The code example shown uses two File objects, one for the
// source and another for the destination.  The path for the
// actual file is entered by the program user for both the
// source and destination files.  See Ch. 11.1 for more
// information about paths and file systems.

File srcFile, dstFile;  // declare the object variables

// First, the user enters the path of the source file, which
// must be a valid file for the loop to end.
boolean repeat = false; // assume a valid path is entered on
                        // first try

do {
    System.out.print("Enter the path of the source file: ");

    // create file object
    srcFile = new File(stdin.readLine());

    // check if source file does NOT exist
    if (!srcFile.exists()) {
        System.out.println("ERROR: file not found");
        repeat = true;
    }
    // check if source file is NOT a file or is NOT readable
    else if (!srcFile.isFile() || !srcFile.canRead()) {
        System.out.println("ERROR: file not readable");
        repeat = true;
    }
    else
        repeat = false;

} while (repeat);
```

```
// Second, the user enters the path of the destination file,
// which must be a valid file for the loop to end.

repeat = false; // assume a valid path is entered on the
                // first try

do {
    System.out.print("Enter the path of the destination” +
                     “ file: ");

    // create the File object
    dstFile = new File(stdin.readLine());

    // if destination file already exists check if it is a
    // directory or is NOT writeable
    if (dstFile.exists() &&
       (dstFile.isDirectory() || !dstFile.canWrite())) {
        System.out.println("ERROR: file not writeable");
        repeat = true;
    }
    else
        repeat = false;

} while (repeat);

// At this point source and destination File objects are
// associate with actual files in the computer’s file
// system.  Next objects are created that enable the program
// to read from or write to the files in a convenient
// manner.

// Reading from a file is done in a manner similar to
// reading from the console, that is, a BufferedReader is
// created.  Recall BufferedReader objects are composed from
// several different objects.  The source of file input is a
// File object. This source must have a mechanism for
// accessing the data, so from it is constructed a
// FileReader object.From this a BufferedReader is
// constructed to more efficiently access the data.

// This is how it is coded:

BufferedReader inFile = new BufferedReader(
                        new FileReader(srcFile));
```

```
// Writing to a file is done in a manner similar to writing
// to the console.  With files a PrintWriter object is
// created, which is composed of several different objects.
// The destination of file output is a File object.  This
// destination file must have a mechanism for writing the
// data, so from it is constructed a FileWriter object.
// From this a BufferedWriter is constructed to more
// efficiently write the data. From this a PrintWriter is
// constructed so that print and prinltln methods can be
// used as done with console I/O.

// This is how it is coded:

PrintWriter outFile = new PrintWriter( new BufferedWriter(
    new FileWriter(dstFile)));

// The inFile and outFile objects can now be used to do file
// I/O.  The following code illustrates some methods that
// are used to read from and write to files.

// The ready method is used to determine if there are more
// characters in the file. It returns true if there are more
// characters in the file, false when the end of the file is
// reached.
while (inFile.ready()) {

    // readLine method reads an entire line (same as console
    // input)
    String line = inFile.readLine();

    // Often a line of characters must be divided into parts
    // (i.e. tokens), which are then converted (i.e. parsed)
    // into the appropriate data types.  Tokenizing a line
    // is done using a StringTokenizer.  StringTokenizer
    // objects are constructed from the string to be divided
    // and a string of delimiters.  Delimiters are the
    // points where the line should be broken.  In this
    // example the string that is read is divided at pipe
    // characters, i.e. "|".
    StringTokenizer tokenizer = new
                            StringTokenizer(line, "|");
```

```
            // If you've created the two line file as instructed
            // above, the first token on a line is a string.  We can
            // get tokens from the StringTokenizer using the
            // nextToken method.  This method returns a String
            // object of the next group of characters upto but not
            // including the next delimiter. Other methods include
            // countTokens that returns the number of tokens left in
            // the StringTokenizer,and hasMoreTokens that returns
            // true if there are more tokens remaining in the
            // StringTokenizer.
            String name = tokenizer.nextToken();

            // Notice the same technique as with console input is
            // used to parse (i.e.convert) the next token into an
            // integer value. Other techniques described in the
            // ConsoleIO.java file can be used to convert to other
            // types of data.
            int year = Integer.parseInt(tokenizer.nextToken());

            // We'll display this line on the console:
            System.out.println("Name: " + name + ", Year: " + year);

            // and also save it in the output file.  PrintWriter
            // objects can do println and print methods that work
            // the same as for System.out.
            outFile.println(name + "|" + year + "|");
        }

        // Finally, always close your files before finishing to
        // guarantee that buffered data isn't lost.
        inFile.close();
        outFile.close();

        // This is how you can read from a file using a
        // Bufferedreader and write to a file using a PrintWriter.
        // By copying the code examples above, you can use file
        // input and output in your programs.
    }
```

Exceptions

- An exception is an object that defines an unusual or erroneous situation that has occurred during run-time

  Examples of exceptions:

  - IOException
  - NumberFormatException
  - NullPointerException
  - IndexOutOfBoundsException

- An exception is *thrown* by a program or the runtime environment and must be handled appropriately. There are two ways to appropriately handle exceptions:

  - catch the exception
  - throw the exception

- There are two kinds of exceptions:

  - Checked exception: the compiler checks to make sure that if a "checked" exception can be thrown, the program has code to handle it appropriately

  - Unchecked exception: the compiler does not check for the handling of unchecked exceptions

- If a method could produce a checked exception, that method should either catch the exception or propagate the exception by including the throws clause to the method header (see below)

- If an exception is not handled appropriately, the program will terminate (abnormally) and produce a message that describes what exception occurred and where in the program it was produced

Catching Exceptions: The **`try`** Statement

- The **`try`** statement identifies a block of statements that may throw an exception
- A **`catch`** clause, which follows a **`try`** block, defines how a particular kind of exception is handled
- A **`try`** block can have zero or more **`catch`** clauses associated with it. Each **`catch`** clause is called an exception handler
- Syntax of the **`try`** statement:

```
try {
    <try statement block>
}

catch (<ExceptionType> <identifier>) {
    <catch statement block>
}

catch ...
```

  - **`try`** & **`catch`** are Java reserved words
  - <**`ExceptionType`**> the class of the thrown exception
  - <**`identifier`**> an instance of the class <**`ExceptionType`**>

- If an exception occurs while executing code inside of a try block, execution immediately jumps to the first exception handler (i.e., catch clause) that matches the thrown exception.
- After executing the statements in the catch clause, control transfers to the statement after the entire try statement

The **`finally`** clause

- an optional set of statements at the end of a **`try`** statement that is executed no matter how the **`try`** block is exited and without regard to if an exception was thrown in the **`try`** statement block
  - if no exception is thrown, the **`finally`** clause will be executed before executing the statements following the **`try`** statement
  - if an exception is thrown, the **`finally`** clause will be executed after the appropriate **`catch`** statement block (if any), but before executing the statements following the **`try`** statement
  - if a **`return`** statement occurs (whether in the **`try`** statement block or the **`catch`** statement block) the **`finally`** clause will execute before returning control to the calling method

- Syntax:

```
try {
    <try statement block>
}

catch (<ExceptionType> <identifier>) {
    <catch statement block>
}

catch ...

}

finally {
    <finally statement block>
}
```

- A finally clause is useful when dealing with files that need to be closed no matter if an exception has occurred or not.

Throwing Exceptions and The **`throws`** clause

- If an exception is thrown and no catch clause applies, control is immediately returned to the method that invoked the method that produced the exception to see if it is caught and handled at that outer level—called propagating the exception.

- Propagation continues until the exception is caught and handled, or until is passed out of the main method, which terminates the program and produces an exception message.

- The propagation of an exception occurs automatically.

- However, if a checked exception is not going to be caught, the method header of the method in which this checked exception may occur must include a **`throws`** clause for the exception

- Syntax:

```
<modifiers> <return type> <name> (<arguments>) throws <ExceptionType> {
     <method body>
}
```

- Example:

```
public static void main (String[] args) throws IOException {
     /* do some stuff with I/O here */
}
```

Example of Exception handling

```
public static int readInt (BufferedReader stdin, String prompt)
    throws IOException {

    int num;

    boolean valid = false;

    do {

        System.out.print (prompt);

        try {

            num = Integer.parseInt (stdin.readLine());

            valid = true;

        }

        catch (NumberFormatException e) {

            // System.out.println (e);

            // e.printStackTrace();

            System.out.println ("Please enter a number.");

        }

    } while (!valid);

    return num;

} // end of readInt
```