Chapter: 6

Exception Handling & I/O

Try, Catch, Throwable, finally, Custom exception, Stream, file I/O & Other I/O

6.1 Exception Handling basics (Recall C/C++ 13.6)

Exception: An exception is an error that occurs at ***run time***. By exception handling subsystem we can handle ***run-time errors*** in a structured and controlled manner.

- Error handling is done by allowing your program to define a block of code, called an ***exception handler***, that is executed automatically when an error occurs.
- Common program errors, such as **divide-by-zero** or **file-not-found** are easily fixed by Exception handling.
- Also, Java's ***API library*** makes extensive use of exceptions.
- Successful Java programmer means that you are fully capable of navigating Java's exception handling subsystem.

❒ ***The Exception Hierarchy:*** In Java, ***all exceptions are represented by classes***. All ***exception classes*** are derived from a ***class*** called ***`Throwable`***. Thus, when an exception occurs in a program, an ***object of some type of exception class is generated***.

- There are two direct subclasses of ***`Throwable`***:
  - ***Error:*** These are the errors that occur in the ***JVM*** itself, and not in program. Program will not usually deal with them.
  - ***Exception:*** Errors that ***result from program*** activity are represented by subclasses of ***Exception***. For example, ***divide-by-zero***, ***array boundary***, and ***file errors*** fall into this category. Program should handle exceptions of these types.
    - ★ An important ***subclass*** of ***Exception*** is ***RuntimeException***, represents various common ***run-time errors***.

❒ ***Keywords for handling exceptions:*** Exception handling is managed via **5 keywords/clause**: ***try***, ***catch***, ***throw***, ***throws***, and ***finally***.

- Program statements that you want to monitor for exceptions are contained within a ***`try`*** block. If an exception occurs within the ***`try`*** block, it is ***thrown***.
- Your code can *catch* that ***thrown*** exception using ***`catch`*** and handle it in some rational manner.
- To manually throw an exception, use the keyword ***`throw`***. (System-generated exceptions are automatically thrown by the ***Java run-time system***.)
- In some cases, an exception that is thrown out of a method must be specified as such by a ***`throws`*** **clause**.
- Any code that absolutely must be executed upon exiting from a ***`try`*** block is put in a ***`finally`*** block.

❒ ***Exceptions are generated in three different ways:***

- ***First***, the ***`Java Virtual Machine`*** can generate an exception in response to some internal error which is beyond your control.
- ***Second***, ***`standard`*** exceptions, such as those corresponding to divide-by-zero or array index out-of-bounds, are generated by errors in program code.
- ***Third***, you can manually generate an exception by using the ***`throw`*** statement.

6.2 try and catch

At the core of exception handling are try and catch. They work together; you can't have a catch without a try. Here is the general form:

```
try { // block of code to monitor for errors }
catch (ExcepType1 exOb) { // handler for ExcepType1 }
catch (ExcepType2 exOb) { // handler for ExcepType2 }
      . . .   . . .  . . .
```

- ***`ExcepType`*** is the type of exception that has occurred.
- Thrown exception is caught by its corresponding ***`catch`*** statement, which then processes the exception.
- There can be more than one ***`catch`*** statement associated with a ***`try`***.
- The type of the exception determines which ***`catch`*** is executed.
- When an exception is caught, ***`exOb`*** will receive its value.

- ***`Catch` statements are executed only if an exception is thrown:*** If no exception is thrown, then a ***`try`*** block ends normally, and all of its ***`catch`*** statements are bypassed. Execution resumes with the first statement following the last ***`catch`***.

❒ ***try-with-resources:*** This form of ***`try`*** is described in Next ***I/O*** Topics, in the context of managing ***I/O streams*** (such as those connected to a file) because streams are some of the most commonly used resources.

Example *(ArrayIndexOutOfBoundsException):* it is an error to attempt to index an array **beyond its boundaries**. When this occurs, the ***JVM*** throws an ***ArrayIndexOutOfBoundsException***. Following program generates such an exception and then catches it:

```
class ExcDemo1{ public static void main(String args[]){ int nums[] = new int[4];
      try {       System.out.println("Before exception");
                  nums[7] = 10;       // Generate an index out-of-bounds exception.
                  System.out.println("this won't be displayed");  }
      catch(ArrayIndexOutOfBoundsException exc) {    // catch the exception
                  System.out.println("Index out-of-bounds"); }
      System.out.println("After catch ");     }}
```

OUTPUT:
Before exception
Index out-of-bounds
After catch

key points about exception handling:

- When an *exception* occurs, the *exception* is thrown out of the ***`try`*** block and caught by the ***`catch`*** statement. At this point, ***control passes*** to the ***`catch`***, and the ***`try`*** block is ***terminated***. [That is, catch is not called. Rather, program execution is transferred to it. Thus, the **println()** statement following the **out-of-bounds index** will *never execute*. After the **catch** statement executes, program control continues with the statements following the **catch**.]

- If no exception is thrown by a ***try*** block, no ***catch*** statements will be executed and control resumes after the ***catch*** statement. Eg In the preceding program, change ***nums[7] = 10;*** to ***nums[0] = 10;*** [Now, no exception is generated, and the catch block is not executed.]

- ***An exception can be generated by one method and caught by another:*** Since all code within a ***try*** block is monitored for exceptions. This ***includes exceptions that might be generated by a method called from within the try block***. An **exception thrown by a method called from within a try block** can be caught by the **catch statements associated with that try block**—assuming, of course, that the method did not catch the exception itself. For example:

```
class ExcTest {
        static void genException() {
                int nums[] = new int[4];
                System.out.println("Before exception.");
                nums[7] = 10;     // Generate an exception.
                System.out.println("Never display");     }
        }
```

```
class ExcDemo2 {
public static void main(String args[]) {
        try { ExcTest.genException(); }
        catch (ArrayIndexOutOfBoundsException exc) {
                // catch the exception
        System.out.println("Index out-of-bounds!"); }
        System.out.println("After catch statement.");    }}
```

  - Since ***genException()*** is called from within a ***try*** block, the exception that it generates (and does not catch) is caught by the catch in ***main()***.

- If ***genException()*** had caught the exception itself, it ***never would have been passed back*** to ***main()***.

NOTE:

[1] ***Catch a JVM exception inside a program to avoid ABNORMAL Termination:*** Catching one of Java's standard exceptions, as the preceding program does, has a side benefit: ***It prevents abnormal program termination***.

- When an exception is thrown, it must be caught by some piece of code, somewhere. If that exception isn't caught, then it will be caught by the JVM. The trouble is that the **JVM's default exception handler *terminates execution*** and displays a stack trace and error message.

[2] ***TYPE specification inside CATCH:*** The ***type*** of the **exception** must match the ***type*** specified in a ***catch***. If it doesn't, the **exception** won't be **caught**. For example, the following tries to catch an ***array boundary error*** with a ***catch*** for an ***ArithmeticException*** (another of Java's built-in exceptions). When the array boundary is overrun, an ***ArrayIndexOutOfBoundsException*** is generated, but it won't be caught by the ***catch***. This results in abnormal program termination.

```
class ExcTypeMismatch { public static void main(String args[]) {
        int nums[] = new int[4];
        try {System.out.println("Before exception is generated.");
                nums[7] = 10;          // generate an index out-of-bounds exception
                System.out.println("this won't be displayed"); }
/* Can't catch an array boundary error with an ArithmeticException. */
        catch(ArithmeticException exc){ System.out.println("Index out-of-
bounds!"); }
        System.out.println("After catch statement."); }}
```

OUTPUT

```
Before exception is generated.

Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 7
at
ExcTypeMismatch.main(ExcTypeMismatch.java:10)
```

[3] ***Exception handling makes a program to respond to an error and then continue running:*** following code divides the elements of one array by the elements of another array. If a ***division by zero*** occurs, an ***ArithmeticException*** is generated. In the program, this exception is handled by reporting the error and then continuing with execution. Preventing run-time-error.

```
class ExcDemo3 { public static void main(String args[]) {
int numer[] = { 4, 8, 16, 32, 64, 128 };
int denom[] = { 2, 0, 4, 4, 0, 8 };
        for(int i=0; i<numer.length; i++) {
                try { System.out.println(numer[i] + " / " + denom[i] + " is " + numer[i]/denom[i]); }
                catch (ArithmeticException exc) { System.out.println("Can't divide by Zero!"); }
        }       }}
```

OUTPUT :

```
4 / 2 is 2
Can't divide by Zero!
16 / 4 is 4
32 / 4 is 8
Can't divide by Zero!
128 / 8 is 16
```

REMEMBER: Once an exception has been handled, it is removed from the system. Therefore, in the program, each pass through the loop enters the try block anew; any prior exceptions have been handled. It handles ***repeated errors***.

6.3 Try and catch advanced

- Multiple-Catch: More than one ***catch*** allowed in a ***try*** if ***each catch must catch a different type*** of ***exception***. Eg:

```
        //Following catches both array boundary and divide-by-zero errors:
class ExcDemo4 { public static void main(String args[]) { int numer[] = { 4, 8, 16, 32, 64, 128, 256, 512 };
                                                           int denom[] = { 2, 0, 4, 4, 0, 8 };
for(int i=0; i<numer.length; i++) {
                try { System.out.println(numer[i] + " / " + denom[i] + " is " + numer[i]/denom[i]); }
/* catches devide by 0 */   catch (ArithmeticException exc) { System.out.println("Can't divide by Zero!"); }
/* catches array boundery */ catch (ArrayIndexOutOfBoundsException exc) { System.out.println("No matching found."); }     }     }}
```

- ***Each catch statement responds only to its own type of exception***. In general, catch expressions are checked in the order in which they occur in a program. Only a matching statement is executed. All other catch blocks are ignored.

- ***Catching Subclass Exceptions:*** A ***catch*** for a ***superclass-Exception-class*** will also match any of its ***subclasses-Exception-class***. [For example, since the **superclass** of all **exceptions** is **Throwable**, to catch all possible exceptions, *catch* **Throwable**.].

  - If you want to ***catch*** **exceptions** of both a ***Super-Exception-Class type*** and a ***subclass type***, put the ***sub-Exception-class*** first in the ***catch sequence***. If you don't, then the ***Super-Exception-Class catch*** will also catch all **derived exception classes**.
  *[This rule is self-enforcing because putting the **superclass first** causes **unreachable code** to be created, since the **subclass catch** can never execute. In Java, **unreachable code** is an **error**.]*

Example: ***Subclasses*** must precede ***superclasses*** in ***catch*** statements.

```
class ExcDemo5 { public static void main(String args[]) {  int numer[] = { 4, 8, 16, 32, 64, 128, 256, 512 };
                                                            int denom[] = { 2, 0, 4, 4, 0, 8 };        // Here, numer is longer than denom.
   for(int i=0; i<numer.length; i++){ try{ System.out.println(numer[i] + " / " + denom[i] + " is " + numer[i]/denom[i]); }
/* catches array boundery */                  catch (ArrayIndexOutOfBoundsException exc){ System.out.println("No matching."); }
/* automatically catch divide by 0 */        catch (Throwable exc) { System.out.println("Other exception"); } } }}
```

☞ In this case, ***catch(Throwable)*** catches all exceptions except for ***ArrayIndexOutOfBounds***-Exception.

☠ ***Why catch superclass exceptions?:*** There are, of course, a variety of reasons. Here are three,

- First, if you add a **catch** that catches exceptions of type **Exception**, then you have effectively added a ***"catch all"*** to your *exception handler* that deals with all program-related exceptions. Such a ***"catch all"*** clause is useful in situation in which abnormal program termination must be avoided no matter what occurs.
- Second, in some situations, an *entire category of exceptions can be handled by the same* ***catch-clause***. Catching the **superclass of these exceptions** allows you to handle all without duplicated code.
- Third, catching subclass exceptions becomes more important when you create exceptions of your own.

❒ ***Nested Try Blocks:*** One ***try block*** can be nested within another. If an **exception** generated within the ***inner try*** remain ***uncaught***, then it could be caught by a ***catch*** associated ***outer try***. For example, here the ***ArrayIndexOutOfBoundsException*** is not caught by the ***inner catch***, but by the ***outer catch***:

```
class NestTrys { public static void main(String args[]) {  int numer[] = { 4, 8, 16, 32, 64, 128, 256, 512 };
                                                            int denom[] = { 2, 0, 4, 4, 0, 8 };
/* outer try */     try { for(int i=0; i<numer.length; i++) {
/* nested try */              try { System.out.println(numer[i] + " / " + denom[i] + " is " + numer[i]/denom[i]); }
/* catch with nested try */   catch (ArithmeticException exc) { System.out.println("Can't divide by Zero!"); } /* a catch must associated with a try */
                              }
                    }
/* outer catch */   catch (ArrayIndexOutOfBoundsException exc) {System.out.println("No matching element found.");}     }}
```

☞ In this example, an exception that can be handled by the ***inner try***—in this case, a ***divideby- zero error***—allows the program to ***continue***. However, an ***array boundary error*** is caught by the ***outer try***, which causes the program to ***terminate***.

💣 Often **nested try blocks** are used to allow different categories of errors to be handled in different ways. Use an ***outer try*** block to catch the most ***severe errors***, allowing ***inner try*** blocks to handle ***less serious*** ones.

6.4 Throw, Rethrow and Subclasses of throwable

❒ ***Throwing an Exception:*** To manually throw an exception by using the throw statement use following general form:

```
throw exceptOb;
```

☛ Here, ***exceptOb*** must be an *object of an exception* class derived from ***Throwable***.

Example: manually throwing an ***ArithmeticException*** using the ***throw*** statement:

```
class ThrowDemo { public static void main(String args[]) {
          try{   System.out.println("Before throw.");
                 throw new ArithmeticException();}        //Throw an exception-type object
          catch(ArithmeticException exc) { System.out.println("Exception caught."); }
          System.out.println("After try/catch block."); }}
```

☛ Notice how the ***ArithmeticException*** was created using ***new*** in the ***throw*** statement.

☛ Also notice ***throw*** is inside of a ***try*** block

☠ Remember, ***throw*** throws an object. Thus, you must create an object for it to ***throw***. That is, you can't just throw a ***type***.

☝ Most often, the exceptions that you will throw will be instances of exception classes that you created. Creating your own exception classes allows you to handle errors in your code as part of your program's overall exception handling strategy.

❒ ***Rethrowing an Exception:*** An *exception* caught by one ***catch*** can be ***rethrown*** so that it can be caught by an outer ***catch***. The most likely reason for ***rethrowing*** this way is to allow *multiple handlers access to the exception*. *(For example, perhaps one exception handler manages one aspect of an exception, and a second handler copes with another aspect.*

☛ Remember, when you ***rethrow*** an *exception*, it will ***not*** be *recaught* by the ***same catch***. It will send to the ***next catch***.

Example: The following program illustrates *rethrowing* an exception:

```
class Rethrow { public static void genException() { int numer[] = { 4, 8, 16, 32, 64, 128, 256, 512 };
                                                    int denom[] = { 2, 0, 4, 4, 0, 8 };
                for(int i=0; i<numer.length; i++) {
                        try { System.out.println(numer[i] + " / " + denom[i] + " is " + numer[i]/denom[i]); }
                        catch (ArithmeticException exc) { System.out.println("Can't divide by Zero!"); }
                        catch (ArrayIndexOutOfBoundsException exc) { System.out.println("No matching element found.");
                                                                     throw exc; }    // rethrow the exception
                        }       }   }

class RethrowDemo { public static void main(String args[]) {
          try { Rethrow.genException(); }
          catch(ArrayIndexOutOfBoundsException exc) {
                        System.out.println("Fatal error - " + "program terminated."); }   }}
```

💣 In this program, *divide-by-zero* errors are handled *locally*, by ***genException()***, but an *array boundary error* is *rethrown*. In this case, it is caught by ***main()***.

❒ ***Sub-classes of Throwable:*** Since, a ***catch*** clause specifies an ***exception type*** and a ***parameter*** (Eg: ***catch(ArrayIndexOutOfBoundsException exc) { }***. Here ***ArrayIndexOutOfBoundsException*** is an ***exception type*** and ***exc*** is a ***parameter***). The parameter receives the exception object.

☞ Since all exceptions are subclasses of Throwable, all exceptions support the methods defined by Throwable. Commonly used Throwable methods are shown in following Table.

Method	***Description***
Throwable fillInStackTrace( )	Returns a Throwable object that contains a completed stack trace. This object can be rethrown.
String getLocalizedMessage( )	Returns a localized description of the exception.
String getMessage( )	Returns a description of the exception.
void printStackTrace( )	Displays the stack trace.
void printStackTrace(PrintStream stream)	Sends the stack trace to the specified stream.
void printStackTrace(PrintWriter stream)	Sends the stack trace to the specified stream.
String toString( )	Returns a String object containing a complete description of the exception. This method is called by println( ) when outputting a Throwable object.

☞ ***printStackTrace( )***: To ***display*** the standard error message plus a record of the method calls that lead up to the exception.

☞ ***toString( )***: To ***retrieve*** the standard error message. The ***toString( )*** method is also called when an exception is used as an argument to ***println( )***.

Example: The following program demonstrates ***printStackTrace( )*** method:

```
class ExcTest {
        static void genException() {
                int nums[] = new int[4];
                System.out.println("Before exception.");
                nums[7] = 10; // out-of-bounds exception
                System.out.println("won't be displayed");     }
        }
```

```
class UseThrowableMethods {
public static void main(String args[]) {
        try { ExcTest.genException(); }
        catch (ArrayIndexOutOfBoundsException exc) {
                System.out.println("Standard message is: ");
                System.out.println(exc);
                System.out.println("\nStack trace: ");
                exc.printStackTrace(); }
        System.out.println("After catch statement."); }}
```

6.5 Finally and Throws

❒ ***Finally:*** Sometimes you will want to define a block of code that will execute when a try/catch block is left (i.e. after a try/catch).
[For example, an exception might cause an error that terminates the current method, causing its premature return. However, that method may have opened a file or a network connection that needs to be closed. Java provides a convenient way to handle these kind of problem: ***finally****]*

☞ To specify a block of code to execute when a ***try/catch*** block is exited, include a ***finally*** block at the end of a ***try/catch sequence***. The general form of a ***try/catch*** that includes ***finally*** is:

```
try { // block of code to monitor for errors }
catch (ExcepType1 exOb) { /* handler for ExcepType1 */ }
catch (ExcepType2 exOb) { /* handler for ExcepType2 */ }
      //...
finally { // finally code }
```

☑ The ***finally*** block will be executed whenever execution leaves a try/catch block, no matter what conditions cause it. That is, ***whether the try block ends normally, or because of an exception, the last code executed is that defined by finally***.

☑ The ***finally*** block is also executed if any code within the ***try*** or any of its ***catch*** statements ***return*** from the ***method***.

Example: Here is an example of finally:

```
class UseFinally {
    public static void genException(int what) {
        int t;
        int nums[] = new int[2];
        System.out.println("Receiving " + what);
        try { switch(what) { case 0: t = 10 / what; break;        /* div-by-zero error */
                             case 1: nums[4] = 4; break;          /* array index error. */
                             case 2: return; /* return from try block */   }
            }
        catch (ArithmeticException exc) { System.out.println("Can't divide by Zero!");
                                          return; /* return from catch. Goes to finally */      }
        catch (ArrayIndexOutOfBoundsException exc) {
                          System.out.println("No matching element found."); }
        finally { System.out.println("Leaving try."); } /* executed on the way out of try/catch*/ }   }
```

```
class FinallyDemo {
public static void main(String args[]) {
    for(int i=0; i < 3; i++) {
        UseFinally.genException(i);
        System.out.println();       }
    }}
```

OUTPUT:

```
Receiving 0
Can't divide by Zero!
Leaving try.
Receiving 1
No matching element found.
Leaving try.
Receiving 2
Leaving try.
```

As the output shows, ***no matter how*** the ***try*** block is exited, the ***finally*** block is ***executed***.

❒ ***Throws:*** In some cases, if a method ***generates an exception*** that ***it does not handle***, it must declare that exception in a ***throws*** clause. Here is the general form of a method that includes a throws clause:

```
ret-type methName(param-list) throws except-list { // body }
```

Here, ***except-list*** is a ***comma-separated list*** of exceptions that the method might throw outside of itself.

NOTE: The reason for ***not specifying*** a ***throws*** clause for ***some of the preceding examples*** is that exceptions that are subclasses of ***Error*** or ***RuntimeException*** don't need to be specified in a ***throws*** list.
[Java simply assumes that a method may throw one. All other types of exceptions do need to be declared. Failure to do so causes a compile-time error.]

☞ ***throws java.io.IOException:*** Actually, you saw an example of a ***throws*** clause earlier in this book. As you will recall (See: Java/C# 1.18), when performing ***keyboard input***, you needed to add the clause ***throws java.io.IOException*** to ***main()***. Now you can understand why:

☺ An ***input statement*** might generate an ***IOException***, and at that time, we weren't able to handle that exception. Thus, such an exception **would be thrown** out of ***main()*** and needed to be specified as such.

Example: Following program handles ***IOException***. It creates a method called ***prompt()***, which displays a *prompting message* and then *reads a character from the keyboard*. Since input is being performed, an ***IOException*** might occur.

☛ However, the ***prompt()*** method does not handle ***IOException*** itself. Instead, it uses a ***throws*** clause, which means that the calling method must handle it. In this example, the calling method is ***main()***, and it deals with the error.

```
class ThrowsDemo {
    public static char prompt(String str) throws java.io.IOException {
        System.out.print(str + ": ");
        return (char) System.in.read();  }
    public static void main(String args[]) {
        char ch;
        try { ch = prompt("Enter a letter"); }
        catch(java.io.IOException exc) { System.out.println("I/O exception.");
            ch = 'X';  }
        System.out.println("You pressed " + ch);  }}
```

💣 NOTICE that ***IOException*** is fully qualified by its package name ***java.io***. Because Java's ***I/O*** system is contained in the ***java.io*** package. Thus, the ***IOException*** is also contained there. It would also have been possible to import ***java.io*** and then ***refer*** to ***IOException directly***.

6.6 Built-in Exceptions and Some Recent Features

❒ ***Java's Built-in Exceptions:*** Inside the standard package ***java.lang***, Java defines several exception classes. The most general of these exceptions are subclasses of the standard type ***RuntimeException***.

☞ ***Unchecked exceptions:*** Since ***java.lang*** is implicitly imported into all Java programs, most exceptions derived from ***RuntimeException*** are automatically available. Furthermore, they ***need not be included in any method's throws list***, these are called ***unchecked exceptions*** because the compiler does not check to see if a method handles or throws these exceptions.

*Table:1 The **Unchecked** Exceptions Defined in **java.lang***

Exception	Meaning
ArithmeticException	Arithmetic error, such as integer divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBoundsException	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

☞ ***Checked exceptions:*** This Table:2 lists those exceptions defined by ***java.lang*** that must be included in a method's throws list if that method can generate one of these exceptions and does not handle it itself. These are called checked exceptions.

*Table:2 The **Checked** Exceptions Defined in **java.lang***

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface
IllegalAccessException	Access to a class is denied
InstantiationException	Attempt to create an object of an abstract class or interface
InterruptedException	One thread has been interrupted by another thread
NoSuchFieldException	A requested field does not exist
NoSuchMethodException	A requested method does not exist
ReflectiveOperationException	Superclass of reflection-related exceptions

☞ **Other exceptions:** In addition to the exceptions in ***java.lang***, Java defines several other types of exceptions that relate to other packages, such as ***IOException*** mentioned earlier.

❒ **Three Recently Added Exception Features:** Beginning with JDK 7, Java's exception handling mechanism has been expanded with the addition of three features:

☞ ***Automatic resource management:*** It automates the process of releasing a resource, such as a file, when it is no longer needed. It is based on an expanded form of ***try***, called the ***try-with-resources statement***, and it is described in ***I/O***.

☞ ***multi-catch:*** *Multi-catch* allows two or more ***exceptions*** to be caught by the same ***catch*** clause. It is not uncommon to have situations in which ***two or more catch clauses execute the same code sequence even though they catch different exceptions***. Instead of having to ***catch*** each exception type individually, you can now use a single ***catch*** clause to handle the ***exceptions*** without ***code duplication***.

🛠 To create a ***multi-catch***, specify a ***list of exceptions*** within a single ***catch*** clause by separating each ***exception type*** in the ***list*** with the ***OR operator***. Each **multi-catch** parameter is implicitly ***final*** [Recall Java/C# 4.7 Final]. Can't give new value.

Example: Following uses the **multi-catch** feature to catch both ***ArithmeticException*** and ***ArrayIndexOutOfBoundsException*** with a single ***catch*** clause: ***catch(final ArithmeticException | ArrayIndexOutOfBoundsException e) {***

```
class MultiCatch { public static void main(String args[]) {  int a=88, b=0;
                                                              int result;
                                                              char chrs[] = { 'A', 'B', 'C' };
```

```
for(int i=0; i < 2; i++) { try { if(i == 0) result = a / b;      // generate an ArithmeticException
                                 else chrs[5] = 'X';  // generate an ArrayIndexOutOfBoundsException
                                 }
                                          // This catch clause catches both exceptions.
                           catch(ArithmeticException | ArrayIndexOutOfBoundsException e) { System.out.println("Err :" + e); }
                           }
System.out.println("After multi-catch.");      }}
```

The program will generate an ***ArithmeticException*** when the ***division by zero*** is attempted. It will generate an ***ArrayIndexOutOfBoundsException*** when the attempt is made to access ***outside the bounds*** of ***chrs***. Both exceptions are caught by the single ***catch*** statement.

final rethrow or more precise rethrow: The ***more precise rethrow*** or ***final rethrow*** feature ***restricts the type of exceptions*** that can be ***rethrown*** to only those **checked exceptions** that the associated ***try*** block throws, that are not handled by a preceding ***catch*** clause, and that are a ***subtype*** or ***supertype*** of the parameter. For the ***final rethrow*** feature to be in force, the ***catch*** **parameter** must be effectively ***final***.

6.7 Chained exceptions

Chained exceptions: The chained exception feature allows you to specify one exception as the underlying cause of another. For example, imagine a situation in which a method throws an **ArithmeticException** because of an attempt to divide by zero. However, the actual cause of the problem was that an I/O error occurred, which caused the divisor to be set improperly.

To allow ***chained exceptions***, two *constructors* and two *methods* were added to ***Throwable*** class.

The ***constructors*** are :
```
Throwable(Throwable causeExc)
Throwable(String msg, Throwable causeExc)
```

- In the first form, ***causeExc*** is the *exception* that causes the *current exception*.
- The second form allows you to specify a *description* at the same time that you specify a *cause exception*.
- These two constructors have also been added to the ***Error***, ***Exception***, and ***RuntimeException classes***.

The chained ***exception methods*** added to ***Throwable*** are ***getCause()*** and ***initCause()***. These methods are shown here:
```
Throwable getCause()
Throwable initCause(Throwable causeExc)
```

- ***getCause()*** returns the exception that underlies the *current exception*. If there is no *underlying exception*, ***null*** returned.
- ***initCause()*** associates ***causeExc*** with the invoking exception and returns a *reference to the exception*. Thus, you can associate a ***cause*** with an exception ***after the exception has been created***. In general, ***initCause()*** is used to set a cause for legacy exception classes that don't support the two additional constructors described earlier.

6.8 Creating Exception Subclasses

Through the use of ***custom exceptions***, you can manage errors that relate specifically to your application. Creating an exception class is easy. Just define a *subclass* of ***Exception*** (which is, of course, a subclass of ***Throwable***).

Your subclasses ***don't need to actually implement anything***—it is their existence in the type system that allows you to use them as exceptions.

The ***Exception*** class does not define any methods of its own. It does, of course, inherit those methods provided by ***Throwable***. Of course, you can override one or more of these methods in ***custom exceptions.***

Example: Following creates an exception called ***NonIntResultException***, which is generated when the result of dividing two **integer** values produces a result with a **fractional** component. ***NonIntResultException*** has two fields which hold the integer values; a *constructor*, and an *override* of the ***toString()*** method, allowing the description of exception to be displayed using ***println()***.

```
class NonIntResultException extends Exception { int n, d;
                                                 NonIntResultException(int i, int j) { n = i; d = j; }
                                                 public String toString() { return "Result of "+ n + " / " + d + " is non-integer."; }
                                                 }

class CustomExceptDemo { public static void main(String args[]) {
    // Here, numer contains some odd values.
    int numer[] = { 4, 8, 15, 32, 64, 127, 256, 512 };
    int denom[] = { 2, 0, 4, 4, 0, 8 };
    for(int i=0; i<numer.length; i++) {
                    try{    if((numer[i]%2) != 0) throw new NonIntResultException(numer[i], denom[i]);
                            System.out.println(numer[i] + " / " + denom[i] + " is " + numer[i]/denom[i]);    }
                    catch (ArithmeticException exc) { System.out.println("Can't divide by Zero!"); }
                    catch (ArrayIndexOutOfBoundsException exc) { System.out.println("No matching element found."); }
                    catch (NonIntResultException exc) { System.out.println(exc); }
    }    }}
```

```
OUTPUT
4 / 2 is 2
Can't divide by Zero!
Result of 15 / 4 is non-integer.
32 / 4 is 8
Can't divide by Zero!
Result of 127 / 8 is non-integer.
No matching element found.
No matching element found.
```

6.9 Java I/O System

Java ***I/O*** system is based upon a hierarchy of classes. Java's ***I/O*** system is quite large, containing many classes, interfaces, and methods. Part of the reason for its size is that Java defines **two complete I/O systems**: one for ***byte I/O*** and the other for ***character I/O***.

NOTE: The **I/O** classes described in this chapter support **text-based console I/O** and **file I/O**. They are not used to create ***graphical user interfaces (GUIs)***. Thus, you will not use them to create *windowed applications*, for example. However, Java does include substantial support for building graphical user interfaces. The basics of ***GUI*** programming are found in ***Chapter 10***, where ***applets*** are introduced; ***Chapter 11***, which offers an introduction to ***Swing***; and an overview of ***JavaFX***. (***Swing*** and ***JavaFX*** are two of Java's ***GUI toolkits***.)

6.10 Byte Streams, Character Streams and Pre-defined Streams

Java perform ***I/O*** through ***streams*** (C/C++ similar). An ***I/O stream*** is an **abstraction** that either *produces* or *consumes* information. Modern versions of Java define ***two types of I/O streams***: ***byte*** and ***character***.

- A **stream** is linked to a **physical device** by the ***Java I/O system***.
- All *streams* behave in the *same* manner, even if the actual ***physical devices they are linked to differ***.

❒ *Byte streams* provide a convenient means for handling *input* and *output* of *bytes*. They are used, for reading or writing binary data and especially helpful when working with files.

- ***Byte Stream Classes***: Byte streams are defined by using *two class hierarchies*. At the top of these are two abstract classes:
- ***InputStream***: **InputStream** defines the characteristics common to *byte input streams*.
- ***OutputStream***: **OutputStream** describes the behavior of *byte output streams*.

From **InputStream** and **OutputStream** are created several concrete *subclasses* that offer varying functionality and handle the details of *reading* and *writing* to various *devices*, such as disk files. The **byte stream classes** are shown in ***Table*** in ***right***.

Byte Stream Class	*Meaning*
`BufferedInputStream`	***Buffered input*** stream
`BufferedOutputStream`	***Buffered output*** stream
`ByteArrayInputStream`	Input stream that ***reads*** from a byte array
`ByteArrayOutputStream`	Output stream that ***writes*** to a byte array
`DataInputStream`	An input stream that contains ***methods for reading*** the Java ***standard data types***
`DataOutputStream`	An output stream that contains ***methods for writing*** the Java ***standard data types***
`FileInputStream`	Input stream that ***reads*** from a ***file***
`FileOutputStream`	Output stream that ***writes*** to a ***file***
`FilterInputStream`	Implements ***InputStream***
`FilterOutputStream`	Implements ***OutputStream***
`InputStream`	***Abstract class*** that describes ***stream input***
`ObjectInputStream`	Input stream for objects
`ObjectOutputStream`	Output stream for objects
`OutputStream`	***Abstract class*** that describes ***stream output***
`PipedInputStream`	Input ***pipe***
`PipedOutputStream`	Output ***pipe***
`PrintStream`	Output stream that contains `print()` and `println()`
`PushbackInputStream`	Input stream that allows ***bytes*** to be ***returned*** to the stream
`SequenceInputStream`	Input stream that is a ***combination*** of two or more ***input streams*** that will be read ***sequentially***, one after the other

❒ *Character streams* are designed for handling the *input* and *output* of *characters*. They use ***Unicode*** and, therefore, can be ***internationalized***. Also, in some cases, ***character streams*** are more efficient than ***byte streams***.

- ***Character Stream Classes:*** Character streams are defined by using *two class hierarchies* topped by these two abstract classes:
- ***Reader:*** **Reader** is used for *input*.
- ***Writer:*** **Writer** is used for *output*.

From `Reader` and `Writer` are derived several concrete *subclasses* that handle various I/O situations (operate on *Unicode character streams*). In general, the ***character-based classes*** parallel the ***byte-based classes***. The character stream classes are shown in Table in Right.

Character Stream Class	*Meaning*
`BufferedReader`	***Buffered*** input ***character stream***
`BufferedWriter`	***Buffered*** output ***character stream***
`CharArrayReader`	***Input*** stream that reads from a ***character array***
`CharArrayWriter`	***Output*** stream that writes to a ***character array***
`FileReader`	***Input*** stream that ***reads*** from a ***file***
`FileWriter`	***Output*** stream that ***writes*** to a ***file***
`FilterReader`	***Filtered*** reader
`FilterWriter`	Filtered ***writer***
`InputStreamReader`	***Input*** stream that **translates** ***bytes*** to ***characters***
`LineNumberReader`	***Input*** stream that ***counts lines***
`OutputStreamWriter`	***Output*** stream that **translates** ***characters*** to ***bytes***
`PipedReader`	***Input*** pipe
`PipedWriter`	Output ***pipe***
`PrintWriter`	***Output*** stream that contains `print()` and `println()`
`PushbackReader`	Input stream that allows ***characters*** to be ***returned*** to the input stream
`Reader`	***Abstract class*** that describes ***character*** stream ***input***
`StringReader`	***Input*** stream that ***reads*** from a ***string***
`StringWriter`	***Output*** stream that ***writes*** to a ***string***
`Writer`	***Abstract*** class that describes ***character*** stream ***output***

[NOTE: For the most part, the functionality of ***byte streams*** is paralleled by that of the ***character streams***. At the lowest level, all I/O is still ***byte-oriented***. The *character-based streams simply provide a convenient and efficient means for handling characters.*]

❒ ***The Predefined Streams:*** All Java programs automatically import the `java.lang` package. This package defines a class called `System`, which encapsulates several aspects of the ***run-time environment***. Among other things, it contains three predefined ***stream variables***, called `in`, `out`, and `err`. These fields are declared as `public`, `final`, and `static` within `System`. I.e. they can be used by any other part of a program and without *reference* to a specific *System* object.

- `System.in` refers to **standard input**, which is by default the *keyboard*. `System.in` is an object of type `InputStream`;
- `System.out` refers to the **standard output stream**. By default, this is the *console*. `System.out` an object of type `PrintStream`.
- `System.err` refers to the **standard error stream**, it is also the *console* by default. `System.err` is also a `PrintStream` type object.
  - However, these streams can be redirected to **any compatible I/O device**.
  - These are ***byte streams***, even though they are typically used to ***read*** and ***write*** characters from and to the console.

6.11 Console I/O using BYTE Streams

As explained, at the top of the ***byte stream*** hierarchy are the ***InputStream*** and ***OutputStream*** classes. ***Table BS-1*** shows the methods in ***InputStream***, and ***Table BS-2*** shows the methods in ***OutputStream***. In general, the methods in InputStream and OutputStream can throw an **IOException** on ***error***. The methods defined by these two **abstract classes** are available to all of their ***subclasses***.

🛠 Recall ***Java/ C# 5.5 Importing package***, ***import java.io.*;*** importing Entire ***java.io*** package

❒ ***Reading Console Input:*** For commercial code, the preferred method of ***reading console input*** is to use a ***character-oriented stream***. Doing so makes your program easier to ***internationalize*** and easier to ***maintain***.

☯ It is more convenient to operate directly on ***characters*** rather than ***converting*** back & forth between ***chars*** and ***bytes***. [However, for sample programs, simple utility programs, and applications that deal with ***raw keyboard input***, using the byte streams is ok.]

Table BS-1: The Methods Defined by InputStream

Method	Description
int available()	Returns the ***number of bytes*** of input ***currently*** available for reading.
void close()	***Closes*** the ***input source***. Further *read attempts* will generate an ***IOException***.
void mark(int numBytes)	Places a ***mark*** at the ***current point*** in the *input stream* that will remain valid until ***numBytes*** bytes are ***read***.
boolean markSupported()	Returns ***true*** if ***mark()/reset()*** are supported by the invoking stream.
int read()	Returns an ***integer representation*** of the next available byte of input. –1 is returned when the end of the stream is encountered.
int read(byte buffer[ ])	Attempts to ***read*** up to ***buffer.length*** bytes into buffer and returns the ***actual number of bytes*** that were successfully read. –1 is returned when the end of the stream is encountered.
int read(byte buffer[ ], int offset, int numBytes)	Attempts to read up to ***numBytes*** bytes into buffer starting at ***buffer[offset]***, returning the number of bytes successfully read. –1 is returned when the end of the stream is encountered.
void reset()	Resets the ***input pointer*** to the previously set mark.
long skip(long numBytes)	Ignores (that is, skips) ***numBytes*** bytes of input, returning the ***number of bytes*** actually ignored.

Table BS-2: The Methods Defined by OutputStream

void close()	***Closes*** the ***output*** stream. Further write attempts will generate an ***IOException***.
void flush()	Causes any ***output*** that has been buffered to be sent to its ***destination***. That is, it ***flushes*** the ***output buffer***.
void write(int b)	Writes a ***single byte*** to an ***output*** stream. Note that the ***parameter*** is an ***int***, which allows you to call ***write( )*** with expressions without having to ***cast*** them back to ***byte***.
void write(byte buffer[ ])	Writes a ***complete array*** of ***bytes*** to an ***output*** stream.
void write(byte buffer[], int offset, int numBytes)	Writes a ***subrange*** of ***numBytes*** bytes from the ***array buffer***, beginning at ***buffer[offset]***.

Example: Following reads an array of bytes from ***System.in***. Notice that any ***I/O*** **exceptions** that might be generated are simply thrown out of ***main()***.

```
// Read an array of bytes from the keyboard.
import java.io.*;   // 5.5 importing Entire package
class ReadBytes {
public static void main(String args[])
throws IOException {
    byte data[] = new byte[10];
    System.out.println("Enter characters.");
    System.in.read(data); // Read an array of
                          // bytes from the keyboard.
    System.out.print("You entered: ");
    for(int i=0; i < data.length; i++)
        System.out.print((char) data[i]);
}}
```

OUTPUT: Enter some characters.
Read Bytes
You entered: Read Bytes

☝ Because ***System.in*** is an instance of ***InputStream***, you automatically have access to the **methods** defined by ***InputStream***. Unfortunately, ***InputStream*** defines only one **input method**, ***read( )***, which reads ***bytes***. There are **three versions** of ***read( ):***

[1] ***int read( ) throws IOException*** : reads a single char from ***keyboard*** (from ***System.in***). Returns **–1** when the **end of the stream** is occur.

[2] ***int read(byte data[ ]) throws IOException*** : reads ***bytes*** from the ***input stream*** and puts them into ***data*** until either the array is full, the ***end of stream*** is reached, or an error occurs. It returns the number of bytes read, or ***–1*** when the ***end of the stream*** is encountered.

[3] ***int read(byte data[ ], int start, int max) throws IOException*** : reads ***input*** into ***data*** beginning at the location specified by ***start***. Up to ***max*** bytes are stored. It returns the number of bytes read, or ***–1*** when the ***end of the stream*** is reached.

☛ All three versions throw an ***IOException*** when an error occurs. When reading from ***System.in***, pressing ***ENTER*** generates an ***end-of-stream*** condition.

❒ ***Writing Console Output:*** For the most portable code, character streams are recommended for console output. Because ***System.out*** is a byte stream, however, byte-based console output is still widely used.

☞ ***Console output*** is most easily **accomplished** with ***print()*** and ***println()***. These **methods** are defined by the class ***PrintStream*** (which is the type of the object referenced by ***System.out***). Even though ***System.out*** is a ***byte stream***, it is still acceptable to use this stream for simple ***console output***.

☞ Since ***PrintStream*** is an output stream derived from ***OutputStream***, it also implements the **low-level method** ***write()***. Thus, it is possible to write to the console by using ***write()***. The simplest form of ***write()*** defined by ***PrintStream*** is:

void write(int byteval)

💣 writes the byte specified by ***byteval*** to the file. Although ***byteval*** is declared as an ***int***, only the low-order 8 bits are written.

Example: Here is a short example that uses ***write()*** to output the character ***X*** followed by a ***new line*** :

```
class WriteDemo { public static void main(String args[]) { int b;        b = 'X';
                                                            System.out.write(b);  // Write a byte to the screen.
                                                            System.out.write('\n');        }}
```

NOTE

[1] You will not often use ***write()*** to perform **console output** (although it might be useful in some situations), since ***print()*** and ***println()*** are substantially easier to use.

[2] ***PrintStream*** supplies two additional **output methods**: ***printf()*** and ***format()***. Both give you detailed control over the precise ***format*** of data that you output. For example, you can specify the ***number of decimal places*** displayed, a ***minimum field width***, or the ***format of a negative value***.

6.12 File I/O using BYTE Streams

In Java, all files are byte-oriented, and Java provides methods to read and write bytes from and to a file. Thus, reading and writing files using byte streams is very common. However, Java allows you to wrap a byte-oriented file stream within a character-based object.

To create a ***byte stream linked to a file***, use ***FileInputStream*** or ***FileOutputStream***. To **open** a file, simply create an **object** of one of these **classes**, specifying the name of the file as an argument to the constructor. *[Once the file is open, you can **read** from or **write** to it.]*

6.12.1 Reading from a File

Opening a file: A file is **opened for input** by creating a ***FileInputStream*** object. Here is a commonly used **constructor**:

```
FileInputStream(String fileName) throws FileNotFoundException
```

- Here, ***fileName*** specifies the name of the file you want to open.
- If the file ***does not exist***, then ***FileNotFoundException*** is thrown. ***FileNotFoundException*** is a subclass of ***IOException***.

Reading a file: To read from a file, use ***read()***. The version that we will use is: `int read() throws IOException`

- Each time it is called, ***read()*** reads a **single byte** from the file and returns it as an integer value. It returns **–1** when the **end of the file** is encountered.
- It throws an ***IOException*** when an error occurs. Notice, ***read()*** is the same as the one used to read from the ***console***.

Closing a file: Close a file by calling ***close()***. Its general form is: `void close() throws IOException`

- Closing a file ***releases the system resources*** allocated to the file, allowing them to be used by another file.
- ***Failure*** to close a file can result in **"memory leaks"** because of unused resources *remaining allocated*.

Example 1: The following program uses ***read()*** to input and display the contents of a text file, the name of which is specified as a **command-line** argument.

- Notice how the ***try/catch*** blocks handle **I/O errors** that might occur.
- To use this program, specify the name of the file that you want to see. For example, to see a file called ***TEST.TXT***, use the **command line**: `java ShowFile TEST.TXT`

```
import java.io.*;
class ShowFile { public static void main(String args[]) { int i;
                                                    FileInputStream fin;
                                                        // First make sure that a file has been specified.
                                                    if(args.length != 1) { System.out.println("Usage: ShowFile File");    return; }

    try {  fin = new FileInputStream(args[0]); }  //open the file using constructor
    catch(FileNotFoundException exc) {   System.out.println("File Not Found");    return; }

    try {      do {        i = fin.read();   // Read from the file
                           if(i != -1) System.out.print((char) i);
               } while(i != -1);     // read bytes until EOF is encountered. When i equals –1, the end of the file has been reached.
       }
    catch(IOException exc) { System.out.println("Error reading file."); }

    try { fin.close(); }
    catch(IOException exc) { System.out.println("Error closing file."); }        }}
```

Notice it **closes** the file stream after the ***try*** block that **reads the file has completed**. However we can call ***close()*** within a ***finally*** block. In this approach, ***all of the methods*** that access the file are contained within a ***try*** block, and the ***finally*** block is used to ***close*** the file. This way, no matter how the ***try*** block terminates, the file is ***closed***.

```
try{   do{ /* reading as before */} while(i != -1); }
catch(IOException exc){ System.out.println("Error Reading File"); }
finally{      try{fin.close();}     //closing
              catch(IOException exc){ System.out.println("Error Closing File"); }
       } // Close file in the finally block on the way out of the try block.
```

One advantage to this approach in general is that if the code that accesses a file terminates because of some ***non-I/O-related exception***, the file is still closed by the ***finally*** block.

Sometimes it's easier to **wrap the portions** of a program that ***open the file and access the file within a single try block*** (rather than separating the two), and then use a ***finally*** block to ***close*** the file. For example,

```
import java.io.*;
class ShowFile { public static void main(String args[]) {
        int i;
        FileInputStream fin=null;  // Here fin is null
                // First make sure that a file has been specified.

        if(args.length != 1) {
                System.out.println("Usage: ShowFile File");
                return; }
```

```
/* The following code opens a file, reads characters until EOF
   is encountered, and then closes the file via a finally block.*/
try {     fin = new FileInputStream(args[0]);
          do { do{ // reading as before } while(i != -1); }
catch(FileNotFoundException exc) { /* Not Found massage */}
catch(IOException exc) { /* Error massage */}
finally { try { if(fin != null) fin.close(); }
          catch(IOException exc) { /* Error Closing massage */}
        }      }}
```

In this approach, notice that ***fin*** is initialized to ***null***. Then, in the ***finally*** block, the file is closed only if ***fin*** is **not null**. This works because ***fin*** will be **non-null** only if the file was ***successfully opened***. Thus, ***close()*** will *not be called if an exception occurs* while opening the file.

Since ***FileNotFoundException*** is a subclass of ***IOException***, we can use a single catch of ***IOException***:

```
try{ /* . . .  as above  . . . */ }
catch(IOException exc) { System.out.println("I/O Error: " + exc); }
finally { /* . . .  as above  . . . */ }
```

*In this approach, any error, including an error opening the file, will simply be handled by the single **catch** statement. Be aware, however, that **it will not be appropriate in cases in which you want to deal separately with a failure to open a file.***

6.12.2 Writing to a File

🛠 ***Opening a file:*** A file is **opened for output** by creating a ***FileOutputStream*** object. Here is two commonly used **constructor**:

```
FileOutputStream(String fileName) throws FileNotFoundException
FileOutputStream(String fileName, boolean append) throws FileNotFoundException
```

- ➲ If the file **cannot be created**, then ***FileNotFoundException*** is thrown.
- ➲ In the ***first*** form, when an output file is opened, any ***preexisting*** file by the **same name is destroyed**.
- ➲ In the ***second*** form, if append is ***true***, then output is **appended** to the **end of the file**. Otherwise, the file is **overwritten**.

🛠 ***Writing a file:*** To write to a file, use the ***write()***. Its simplest form: ***void write(int byteval) throws IOException***

- ➢ It writes byte specified by ***byteval*** to the file. Although ***byteval*** is declared as an ***int***, only **low-order 8 bits** are written to the file.
- ➢ If an **error** occurs during writing, an ***IOException*** is thrown.

🛠 ***Closing a file:*** *Similar to* ***6.12.1.*** Also ensures that any output remaining in **output buffer** is actually **written** to the **physical device**.

Example: The following example copies a text file. The names of the source and destination files are specified on the command line.

- ◘ To use this program, specify the name of the **source** file and the **destination** file. For example, to copy a file called FIRST.TXT to a file called SECOND.TXT, use the command line: ***java CopyFile FIRST.TXT SECOND.TXT***

```
import java.io.*;

class CopyFile {
public static void main(String args[]) throws IOException {
        int i;
        FileInputStream fin = null;
        FileOutputStream fout = null;

        // First, make sure that both files has been specified.
        if(args.length != 2) {
                System.out.println("Usage: CopyFile from to");
                return;}
```

```
try { // Attempt to open the files.
        fin = new FileInputStream(args[0]);
        fout = new FileOutputStream(args[1]);
        do {    i = fin.read(); // Read bytes from one file
                if(i != -1) fout.write(i); // and write them to another.
            } while(i != -1);    }
catch(IOException exc) { System.out.println("I/O Error: " + exc); }
finally { try { if(fin != null) fin.close();          }
        catch(IOException exc) {
                System.out.println("Error Closing Input File"); }

        try {if(fout != null) fout.close(); }
        catch(IOException exc) {
                System.out.println("Error Closing Output File"); }
      } }}
```

6.12.3 Automatically Closing a File

The automatic closing process is based on another version of the try statement called ***try-with-resources***, and is sometimes referred to as **automatic resource management**.

☯ Main advantage of ***try-with-resources*** is that it prevents situations in which a file (or other resource) is ***inadvertently not released after it is no longer needed***. As explained, forgetting to close a file can result in memory leaks and could lead to other problems.

☛ The try-with-resources statement has this general form: ***try(resource-specification){*** *// use the resource* ***}***

- ➲ Here, ***resource-specification*** is a statement that ***declares*** and ***initializes*** a **resource**, *such as a file*. It consists of a ***variable declaration*** in which the variable is ***initialized*** with a ***reference*** to the *object being managed*.
- ➲ When the ***try*** block ends, the **resource is automatically released**. In the case of a file, this means that the file is automatically closed. (Thus, there is no need to call ***close()*** explicitly.)

💀 A ***try-with-resources*** statement can also include ***catch*** and ***finally*** clauses.

☛ The ***try-with-resources*** statement can be used only with those resources that implement the ***AutoCloseable*** interface defined by ***java.lang***.

- ➲ This interface defines the ***close()*** method. ***AutoCloseable*** is inherited by the ***Closeable*** interface defined in ***java.io***. Both interfaces are implemented by the **stream classes**, including ***FileInputStream*** and ***FileOutputStream***. Thus, ***try-with-resources*** can be used when working with ***streams***, including ***file streams***.

Example 1: As a first example of automatically closing a file, here is a reworked version of the ShowFile program that uses it:

```
import java.io.*;
class ShowFile { public static void main(String args[]) {
                        int i;
                        if(args.length != 1) { System.out.println("Usage: ShowFile filename");      return; }

                /* The following code uses try-with-resources to open a file and then automatically close it when the try block is left. */
/* A try-with-resources block */    try(FileInputStream fin = new FileInputStream(args[0])) { do {    i = fin.read();
                                                                                                    if(i != -1) System.out.print((char) i);
                                                                                                } while(i != -1);    }
                        catch(IOException exc) { System.out.println("I/O Error: " + exc); }
                }}
```

☞ Pay special attention to how the file is opened within the ***try-with-resources*** statement:

try(FileInputStream fin = new FileInputStream(args[0])) { *//……* ***}***

- ➲ Notice how the ***"resource-specification"*** portion of the ***try*** declares a ***FileInputStream*** called ***fin***, which is then assigned a reference to the file opened by its **constructor**.
- ➲ Thus, the variable ***fin*** is local to the ***try***, created when the ***try*** is entered. When the ***try*** is exited, the file associated with ***fin*** is automatically closed by an implicit call to ***close()***.
  - ☠ Remember, the **resource** declared in the ***try*** statement is **implicitly *final***.
  - ☠ Also, the ***scope*** of the ***resource*** is limited to the ***try-with-resources*** statement.

☯ To manage ***more than one resource*** within a single ***try*** statement, simply **separate** each ***resource specification*** with a ***semicolon***.

Example 2: Recall ***CopyFile*** program shown ***6.12.2*** so that it uses a single ***try-with-resources*** statement to manage both ***fin*** and ***fout***.

```
import java.io.*;
class CopyFile { public static void main(String args[]) throws IOException { int i;
        if(args.length != 2) { System.out.println("Usage: CopyFile from to"); return; }

    // Open and manage two files via the try statement.
        try (   FileInputStream fin = new FileInputStream(args[0]);
/* Manage two resources */   FileOutputStream fout = new FileOutputStream(args[1])  ) {   do {   i = fin.read();
                                                                                              if(i != -1) fout.write(i);
                                                                                          } while(i != -1);   }
        catch(IOException exc) { System.out.println("I/O Error: " + exc); } }}
```

☞ In this program, notice how the ***input*** **and** ***output*** **files are opened** within the try:

```
try (  FileInputStream fin = new FileInputStream(args[0]);
       FileOutputStream fout = new FileOutputStream(args[1]) ){  }
```

After this ***try*** block ends, both ***fin*** and ***fout*** will have been ***closed***.

[It is much shorter than the previous version. The ability to ***streamline source code*** is a side-benefit of ***try-with-resources***.]

☯ ***Suppressed exceptions of try-with-resources:*** of In general, when a ***try*** block executes, it is possible that an exception inside the ***try*** block will lead to *another exception that occurs when the resource* is closed in a ***finally*** clause. In the case of a **"normal"** ***try*** statement, the ***original exception is lost***, being preempted by the second exception.

💣 However, with a ***try-with-resources*** statement, the ***second exception is suppressed***. It is ***not***, however, ***lost***. Instead, it is added to the ***list of suppressed exceptions*** associated with the first exception. The list of suppressed exceptions can be obtained by use of the ***getSuppressed()*** method defined by ***Throwable***.

11.13 Reading and Writing Binary Data

To read and write *binary* values of the Java *primitive* types (*int*, *double* or *short*), you will use ***DataInputStream*** and ***DataOutputStream***.

❒ Commonly used *output/input methods* for Java's primitive types are in following Table. Each throws an ***IOException*** on failure.

Output Method	*Purpose*	*Input Method*	*Purpose*
void writeBoolean(boolean val)	Writes the ***boolean*** specified by **val**.	*boolean readBoolean( )*	Reads a ***boolean***.
void writeByte(int val)	Writes the ***low-order byte*** specified by **val**.	*byte readByte( )*	Reads a ***byte***.
void writeChar(int val)	Writes the ***value*** specified by **val** as a *char*.	*char readChar( )*	Reads a ***char***.
void writeDouble(double val)	Writes the ***double*** specified by **val**.	*double readDouble( )*	Reads a ***double***.
void writeFloat(float val)	Writes the ***float*** specified by **val**.	*float readFloat( )*	Reads a ***float***.
void writeInt(int val)	Writes the ***int*** specified by **val**.	*int readInt( )*	Reads an ***int***.
void writeLong(long val)	Writes the ***long*** specified by **val**.	*long readLong( )*	Reads a ***long***.
void writeShort(int val)	Writes the ***value*** specified by **val** as a *short*	*short readShort( )*	Reads a ***short***.

❒ ***DataOutputStream:*** ***DataOutputStream*** implements the ***DataOutput*** interface. This interface defines methods that write all of Java's primitive types to a file. It is important to understand that this data is written using its ***internal, binary format***, not its human-readable ***text form***.

☞ Here is the ***constructor*** for ***DataOutputStream***. Notice that it is built upon an ***instance*** of ***OutputStream***.

DataOutputStream(OutputStream outputStream)

- Here, ***outputStream*** is the stream to which data is written.
- To write output to a file, you can use the object created by ***FileOutputStream*** for this parameter.

❒ ***DataInputStream:*** ***DataInputStream*** implements the ***DataInput*** interface, which provides methods for reading all of Java's primitive types. (See above Table). Remember, ***DataInputStream*** reads data in its ***binary*** format, not its human-readable form.

☞ The ***constructor*** for ***DataInputStream***. Notice that it is built upon an ***instance*** of ***InputStream***.

DataInputStream(InputStream inputStream)

- Here, ***inputStream*** is the stream that is linked to the instance of ***DataInputStream*** being created.
- To read input from a file, you can use the object created by ***FileInputStream*** for this parameter.

Example 1: Here is a program that demonstrates ***DataOutputStream*** and ***DataInputStream***. It writes and then reads back various types of data to and from a file.

```
import java.io.*;
class RWData { public static void main(String args[]) { int i = 10;    double d = 1023.56;    boolean b = true;
// Write some values.
        try (DataOutputStream dataOut = new DataOutputStream(new FileOutputStream("testdata"))) {
                System.out.println("Writing " + i);   dataOut.writeInt(i);          //write binary data
                System.out.println("Writing " + d);   dataOut.writeDouble(d);       //write binary data
                System.out.println("Writing " + b);   dataOut.writeBoolean(b);      //write binary data
                System.out.println("Writing " + 12.2 * 7.4);   dataOut.writeDouble(12.2 * 7.4);    }
        catch(IOException exc) { System.out.println("Write error."); return; }
        System.out.println();
// Now, read them back.
        try (DataInputStream dataIn = new DataInputStream(new FileInputStream("testdata"))) {
                i = dataIn.readInt();       System.out.println("Reading " + i);    // Read binary data.
                d = dataIn.readDouble();    System.out.println("Reading " + d);    // Read binary data.
                b = dataIn.readBoolean();   System.out.println("Reading " + b);    // Read binary data.
                d = dataIn.readDouble();    System.out.println("Reading " + d);                }
        catch(IOException exc) { System.out.println("Read error."); }    }}
```

OUTPUT

```
Writing 10
Writing 1023.56
Writing true
Writing 90.28
Reading 10
Reading 1023.56
Reading true
Reading 90.28
```

Example 2 (A File Comparison Utility): This project develops a simple, yet useful file comparison utility. It works by opening both files to be compared and then reading and comparing each corresponding set of bytes.

- If a ***mismatch*** is found, the files ***differ***.
- If the ***end of each file*** is reached at the same time and if no mismatches have been found, then the files are the ***same***.
- Notice that it uses a ***try-with-resources*** statement to ***automatically close*** the files.

```
import java.io.*;
class CompFiles { public static void main(String args[]){          int i=0, j=0;
    if(args.length !=2 ) { System.out.println("Usage: CompFiles f1 f2");    return; }
// Compare the files.
    try ( FileInputStream f1 = new FileInputStream(args[0]); FileInputStream f2 = new FileInputStream(args[1])  ) {
                do { i = f1.read();     j = f2.read();     if(i != j) break; } while(i != -1 && j != -1);
                if(i != j) System.out.println("Files differ.");
                else System.out.println("Files are the same.");
                }
    catch(IOException exc) { System.out.println("I/O Error: " + exc); }          }}
```

To try ***CompFiles***, first copy ***CompFiles.java*** to a file called ***temp***. Then, try this command line:

```
java CompFiles CompFiles.java temp
```

The program will report that the files are the same.

Next, compare ***CompFiles.java*** to ***CopyFile.java*** (shown earlier) using this command line:

```
java CompFiles CompFiles.java CopyFile.java
```

These files differ and ***CompFiles*** will report this fact.

11.14 Random-Access Files

For random access to a file use ***RandomAccessFile***, which encapsulates a **random-access file**. ***RandomAccessFile*** is not derived from ***InputStream*** or ***OutputStream***. Instead, it implements the **interfaces** ***DataInput*** and ***DataOutput***, which *define the basic I/O methods.*

- It also supports positioning requests—that is, you can position the file pointer within the file. The constructor is :

```
RandomAccessFile(String fileName, String access) throws FileNotFoundException
```

  - Here, the name of the file is passed in ***fileName***
  - ***access*** determines what type of file access is permitted. If it is "***r***", the file can be read but not written. If it is "***rw***", the file is opened in ***read-write mode***.
- To set the current position of the file pointer within the file use the method ***seek()***:

```
void seek(long newPos) throws IOException
```

  - Here, ***newPos*** specifies the **new position**, in bytes, of the ***file pointer*** from the beginning of the file.
  - After a call to ***seek()***, the next read or write operation will occur at the new file position.
- ***RandomAccessFile*** implements ***read()*** and ***write()*** methods. It also implements the ***DataInput*** and ***DataOuput*** **interfaces**, which means that ***methods to read and write the primitive types***, such as ***readInt()*** and ***writeDouble()***, are available.

Example: Following writes six ***doubles*** to a file and then reads them back in ***nonsequential*** (i.e. random-access) order.

```
import java.io.*;
class RandomAccessDemo { public static void main(String args[]) {      double data[] = { 19.4, 10.1, 123.54, 33.0, 87.9, 74.25 };
                                                                        double d;
// Open and use a random access file.
    try (RandomAccessFile raf = new RandomAccessFile("random.dat", "rw")) {          // Open random-access file.
            for(int i=0; i < data.length; i++) { raf.writeDouble(data[i]); }

/* read back specific values */     raf.seek(0);          // seek to first double. Use seek( ) to set the file pointer.
            d = raf.readDouble(); System.out.println("First value is " + d);

            raf.seek(8);          // seek to second double
            d = raf.readDouble(); System.out.println("Second value is " + d);

            raf.seek(8 * 3);      // seek to fourth double
            d = raf.readDouble(); System.out.println("Fourth value is " + d);
            System.out.println();
/* read every other value. */
            System.out.println("Here is every other value: ");
            for(int i=0; i < data.length; i+=2) {    raf.seek(8 * i);      // seek to ith double
                                                     d = raf.readDouble();    System.out.print(d + " ");     }
            }
    catch(IOException exc) { System.out.println("I/O Error: " + exc); }                  }}
```

Notice how each value is located. Since each double value is ***8 bytes*** long, each value starts on an ***8-byte boundary***. Thus, the first value is located at ***zero***, the second begins at ***byte 8***, the third starts at ***byte 16***, and so on. Thus, to read the fourth value, the program seeks to location ***24***.

11.15 Console-based I/O using Console class

Console is primarily a convenience class because most of its functionality is available through ***System.in*** and ***System.out***. However, its use can simplify some types of console interactions, especially when reading strings from the **console**.

Console supplies ***no constructors***. Instead, a Console object is obtained by calling ***System.console()***. It is shown here.

```
static Console console( )
```

- If a console is available, then a ***reference*** to it is returned. Otherwise, ***null*** is returned.
- A console may not be available in all cases, (eg: program runs background). So, if ***null*** is returned, **no console I/O is possible**.

- Console defines several methods that perform I/O, such as ***readLine()*** and ***printf()***.
- It also defines a method called ***readPassword()***, used to obtain a password, without **echoing** what is typed.
- You can also obtain a **reference** to the ***Reader*** and the ***Writer*** that are attached to the console.

6.16 Console I/O Using Character Streams

At the top of the ***character stream*** hierarchy are the **abstract** classes ***Reader*** and ***Writer***. *Table CS-1* shows the methods in ***Reader***, and *Table CS-2* shows the methods in ***Writer***. Most of the methods can throw an **IOException** on **error**. The methods defined by these two abstract classes are available to all of their ***subclasses***.

*Table CS-1: The Methods Defined by **Reader***	
Method	***Description***
abstract void close( )	*Closes the input source.* Further *read attempts* will generate an **IOException**.
void mark(int numChars)	*Places a mark at the current point* in the input stream that will remain valid until **numChars** characters are read.
boolean markSupported( )	Returns **true** if **mark( )/reset( )** are supported on this stream.
int read( )	*Returns an integer representation* of the next available character from the invoking input stream. **–1** is returned when the **end of the stream** is encountered.
int read(char buffer[ ])	*Attempts to read* up to **buffer.length** characters into **buffer** and *returns the actual number of characters* that were successfully read. **–1** is returned when the **end of the stream** is encountered.
abstract int read(char buffer[ ], int offset, int numChars)	*Attempts to read* up to **numChars** characters into buffer starting at **buffer[offset]**, *returning the number of characters* successfully read. **–1** is returned when the **end of the stream** is encountered.
int read(CharBuffer buffer)	*Attempts to fill* the buffer specified by **buffer**, *returning the number of characters* successfully read. **–1** is returned when the **end of the stream** is encountered. **CharBuffer** is a class that encapsulates a sequence of characters, such as a **string**.
boolean ready( )	Returns **true** if the *next input request will not wait.* Otherwise, it returns **false**.
void reset( )	*Resets the input pointer* to the previously set mark.
long skip(long numChars)	Skips over **numChars** characters of input, *returning the number of characters actually skipped.*

*Table CS-2: The Methods Defined by **Writer***	
Method	***Description***
Writer append(char ch)	*Appends* **ch** to the *end of the invoking* output stream. *Returns a reference* to the invoking stream.
Writer append(CharSequence chars)	*Appends* **chars** to the *end of the invoking* output stream. *Returns a reference* to the invoking stream. **CharSequence** is an **interface** that defines **read-only** operations on a sequence of characters.
Writer append(CharSequence chars, int begin, int end)	*Appends* the sequence of **chars** starting at **begin** and stopping with **end** to the end of the invoking output stream. *Returns a reference* to the invoking stream. **CharSequence** is an interface that defines **read-only** operations on a sequence of characters.
abstract void close( )	*Closes the output stream.* Further write attempts will generate an IOException.
abstract void flush( )	*Causes any output that has been buffered to be sent to its destination.* That is, it **flushes** the output buffer.
void write(int ch)	Writes a *single character* to the invoking output stream. Note, the parameter is an **int**, which allows you to call **write( )** with expressions without having to ***cast them back*** to **char**.
void write(char buffer[ ])	Writes a *complete array of characters* to the invoking output stream.
abstract void write(char buffer[ ], int offset, int numChars)	Writes a *subrange* of ***numChars*** characters from the array buffer, beginning at ***buffer[offset]*** to the invoking output stream.
void write(String str)	Writes **str** to the invoking output stream.
void write(String str, int offset, int numChars)	Writes a *subrange* of ***numChars*** characters from the array ***str***, beginning at the specified ***offset***.

❒ ***Constructing BufferedReader :*** Since ***System.in*** is a **byte stream**, you will need to wrap ***System.in*** inside some type of **Reader**. The best class for reading console input is ***BufferedReader***, which supports a **buffered input stream**.

- You cannot construct a ***BufferedReader*** directly from ***System.in***. Instead, you must first convert it into a **character stream**. To do this, you will use ***InputStreamReader***, which **converts bytes to characters**. To obtain an ***InputStreamReader*** object that is linked to ***System.in***, use the constructor: ***InputStreamReader(InputStream inputStream)***
  - ▶ Since ***System.in*** refers to an object of type ***InputStream***, it can be used for **inputStream**.
  - ▶ Next, using the object produced by ***InputStreamReader***, construct a ***BufferedReader*** using the constructor:
    BufferedReader(Reader inputReader)
    - Here, ***inputReader*** is the stream that is linked to the instance of ***BufferedReader*** being created.
    - Putting it all together, the following line of code creates a ***BufferedReader*** that is connected to the **keyboard**.

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

After this statement executes, ***br*** will be a character-based stream that is linked to the console through ***System.in***.

6.16.1 Reading Characters

Characters can be read from ***System.in*** using the ***read()*** method defined by ***BufferedReader*** in much the same way as they were read using ***byte streams***. Here are three versions of ***read()*** supported by ***BufferedReader***.

[1] ***int read() throws IOException*** : reads a single Unicode character. Returns **–1** when the **end of the stream** is occur.

[2] ***int read(char data[ ]) throws IOException*** : reads ***characters*** from the ***input stream*** and puts them into ***data*** until either the array is full, the ***end of stream*** is reached, or an error occurs. It returns the number of ***characters*** read, or ***–1*** when the ***end of the stream*** is encountered.

[3] ***int read(char data[ ], int start, int max) throws IOException*** : reads ***input*** into ***data*** beginning at the location specified by ***start***. Up to ***max characters*** are stored. It returns the number of ***characters*** read, or ***–1*** when the ***end of the stream*** is reached.

- All three versions throw an ***IOException*** when an error occurs. When reading from ***System.in***, pressing ***ENTER*** generates an ***end-of-stream*** condition.

Example: The following program demonstrates ***read()*** by reading characters from the console until the user types a **period**.

☞ Notice that any **I/O exceptions** that might be generated are simply thrown out of ***main()***.

```
import java.io.*;
class ReadChars { public static void main(String args[]) throws IOException {
  char c;
                /* Create BufferedReader linked to System.in. */
  BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
System.out.println("Enter chars, period to quit.");
    do {  c = (char) br.read();   // read characters
          System.out.println(c);
        } while(c != '.');
}}
```

6.16.2 Reading Strings

To read a **string** from the **keyboard**, use the version of readLine( ) that is a member of the BufferedReader class. Its general form is:

```
String readLine( ) throws IOException
```

☞ Returns a ***String object*** that contains the **characters read** & returns ***null*** if an attempt is made to read at the **end of the stream**.

Example: The following program demonstrates ***BufferedReader*** and the ***readLine()*** method. The program reads and displays lines of text until you enter the word "***stop***".

```
import java.io.*;
class ReadLine { public static void main(String args[]) throws IOException {
// create a BufferedReader using System.in
  BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
  String str;
System.out.println("Enter text. Enter 'stop' to quit.");
  //Use readLine( ) from BufferedReader to read a line of text.
    do {      str = br.readLine();
              System.out.println(str);
      } while(!str.equals("stop")); }}
```

6.16.3 Console Output/writing Using Character Streams

It is permissible to use ***System.out*** to write to the console under Java, its use is mostly for *debugging purposes* or for *sample programs*.

- For real-world programs, the ***preferred method of writing to the console*** when using Java is through a ***PrintWriter*** stream. ***PrintWriter*** is one of the **character-based classes**. (It easier to internationalize a program).
  - ☞ ***Constructor for PrintWriter:*** PrintWriter defines several constructors. The one we will use is:

    ```
    PrintWriter(OutputStream outputStream, boolean flushingOn)
    ```

    - ▶ Here, **outputStream** is an object of type ***OutputStream***
    - ▶ ***flushingOn*** controls whether Java ***flushes the output stream*** every time a ***println()*** method (among others) is called. If ***flushingOn*** is ***true***, flushing a*utomatically takes place*. If ***false***, flushing is *not automatic*.
- ***PrintWriter*** supports the ***print()*** and ***println()*** for all types including **Object**. (Use these methods same as ***System.out***). If an argument is not a ***primitive type***, the ***PrintWriter*** will call the object's ***toString()*** method and then print out the result.
  - ☞ To write to the console using a ***PrintWriter***, specify ***System.out*** for the output stream and flush the stream after each call to ***println()***. For example, this line of code creates a ***PrintWriter*** that is connected to console output.

    ```
    PrintWriter pw = new PrintWriter(System.out, true);
    ```

Example: The following application illustrates using a ***PrintWriter*** to handle ***console output***.

```
import java.io.*;
public class PrintWriterDemo { public static void main(String args[]) {
                PrintWriter pw = new PrintWriter(System.out, true);
    // Create a PrintWriter linked to System.out. true means auto-flush is on
                int i = 10;
                double d = 123.65;
pw.println("Using a PrintWriter.");
pw.println(i);
pw.println(d);
pw.println(i + " + " + d + " is " + (i+d)); }}
```

- Remember that there is nothing wrong with using ***System.out*** to write simple text output to the console. However, using a ***PrintWriter*** will make your **real-world applications easier to *internationalize***.

6.17 File I/O Using Character Streams

To perform ***character-based file I/O*** (eg: to store Unicode text), you will use the ***FileReader*** and ***FileWriter*** classes.

- ***Using a FileWriter :*** ***FileWriter*** creates a ***Writer*** that you can use to write to a file. Two commonly used ***constructors*** are:

  ```
  FileWriter(String fileName) throws IOException
  FileWriter(String fileName, boolean append) throws IOException
  ```

  - ☞ Here, **fileName** is the full path name of a file. If append is ***true***, then output is appended to the ***end of the file***. Otherwise, the file is ***overwritten***. Either throws an ***IOException*** on failure.
  - ☞ ***FileWriter*** is derived from ***OutputStreamWriter*** and ***Writer***. So, it has access to the methods defined by these classes.

Example: Following reads lines of text entered at the keyboard and writes them to a file called "***test.txt***". until "***stop***" is entered.

```
import java.io.*;
class KtoD { public static void main(String args[]) { String str;
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Enter text ('stop' to quit).");
        try (FileWriter fw = new FileWriter("test.txt")) { do {  System.out.print(": ");
                                                                 str = br.readLine();
            /* create a file writer */                           if(str.compareTo("stop") == 0) break;
                                                                 str = str + "\r\n";          // add newline
            /* Write strings to the file */                      fw.write(str);     } while(str.compareTo("stop") != 0);
                                                    }
        catch(IOException exc) {System.out.println("I/O Error: " + exc);          }          }}
```

❒ ***Using a FileReader:*** The ***FileReader*** class creates a ***Reader*** that used to read the contents of a file. Commonly used constructor :

FileReader(String fileName) throws FileNotFoundException

- ▶ Here, ***fileName*** is the *full path name of a file.* It throws a ***FileNotFoundException*** if the file does not exist.
- ▶ ***FileReader*** is derived from ***InputStreamReader*** and ***Reader***. So, it has access to the methods defined by these classes.

Example: Following creates a simple **disk-to-screen** utility that reads a text file called "***test.txt***" and displays its contents on the screen. Thus, it is the complement of the **key-to-disk** utility "**KtoD**" shown previously

```
import java.io.*;
class DtoS { public static void main(String args[]) {    String s;
// Create and use a FileReader wrapped in a BufferedReader.
        try (BufferedReader br = new BufferedReader(new FileReader("test.txt"))) {      /* Create a File Reader */
/* Read lines from the file and display them on the screen */          while((s = br.readLine()) != null) { System.out.println(s); }      }
        catch(IOException exc) { System.out.println("I/O Error: " + exc); }        }}
```

☝ In this example, notice that the ***FileReader*** is wrapped in a ***BufferedReader***. This gives it access to ***readLine()***. Also, closing the ***BufferedReader***, ***br*** in this case, *automatically closes the file*.

NOTE

I/O package NIO: Originally called ***New I/O***, ***NIO*** supports a *channel-based* approach to *I/O operations*. The ***NIO*** classes are contained ***in java.nio*** and its subordinate packages, such as ***java.nio.channels*** and ***java.nio.charset***.

☯ NIO is built on two foundational items:

- ➲ ***buffers*** : A ***buffer*** holds data
- ➲ ***channels*** : A ***channel*** represents an *open connection* to an I/O device, such as a **file** or a **socket**.

🛠 To use the ***new I/O*** system, you obtain a ***channel*** to an ***I/O device*** and a ***buffer*** to ***hold*** data. You then operate on the buffer, inputting or outputting data as needed.

☯ Two other entities used by ***NIO*** are:

- ➲ ***charsets:*** A ***charset*** defines the way that *bytes are* ***mapped*** *to characters*. You can ***encode*** a *sequence of characters into bytes* using an ***encoder***. You can ***decode*** a *sequence of bytes into characters* using a ***decoder***.
- ➲ ***selectors:*** A ***selector*** supports ***key-based***, ***non-blocking***, ***multiplexed*** I/O. In other words, selectors enable you to **perform I/O through multiple channels**. ***Selectors*** are most applicable to ***socket-backed channels***.

☝ Beginning with ***JDK 7***, ***NIO*** was enhanced by including three new packages (***java.nio.file***, ***java.nio.file.attribute***, and ***java.nio.file.spi***); several new classes, interfaces, and methods; and direct support for stream-based I/O.

6.18 TYPE WRAPPERS and SCANNER class to convert numeric strings

We know ***println()*** automatically converts numeric values (***int***, ***float*** etc.) into their human-readable form. However, methods like ***read()*** do not reads and converts a string containing a numeric value into its internal, binary format. Eg: there is no version of ***read()*** that reads a string such as ***"100"*** and then automatically converts it into its corresponding binary value that is able to be stored in an ***int*** variable. Instead, Java provides various other ways to accomplish this task. Java's ***type wrappers*** is one of the easiest one.

❒ Java's type wrappers are classes that **encapsulate**, or **wrap**, the ***primitive types***. **Type wrappers** are needed because the **primitive types** are not *objects*, a primitive type cannot be **passed by reference**. To resolve this problem, Java provides classes that correspond to each of the primitive types; these classes are called ***type wrappers***.

☝ The type wrappers are ***Double***, ***Float***, ***Long***, ***Integer***, ***Short***, ***Byte***, ***Character***, and ***Boolean***. These classes offer a wide array of methods that allow you to fully integrate the primitive types into Java's object hierarchy. As a side benefit, the numeric wrappers also define methods that convert a numeric string into its corresponding binary equivalent.

Some conversion methods for ***type wrappers****: Each returns a binary value that corresponds to the string.*

Wrapper	***Conversion Method***
Double	**static double** parseDouble(**String** str) ***throws NumberFormatException***
Float	**static float** parseFloat(**String** str) ***throws NumberFormatException***
Long	**static long** parseLong(**String** str) ***throws NumberFormatException***
Integer	**static int** parseInt(**String** str) ***throws NumberFormatException***
Short	**static short** parseShort(**String** str) ***throws NumberFormatException***
Byte	**static byte** parseByte(**String** str) ***throws NumberFormatException***

☞ The **parsing methods** give us an easy way to ***convert a numeric value***, read as a **string** from the **keyboard** or a **text file**, into its proper internal format.

☞ The **primitive type wrappers** provide a number of ***methods*** that help ***integrate the primitive types into the object hierarchy***. Eg: various **storage mechanisms** provided by the ***Java library***, including ***maps***, ***lists***, and ***sets***, work only with objects. Thus, to store an ***int***, for example, in a list, it must be wrapped in an object.

- ➲ All type wrappers have a **method** called :
  - ✓ ***compareTo()***, which **compares** the value contained within the **wrapper**
  - ✓ ***equals()***, which tests two values for **equality**;
  - ✓ Methods that ***return*** the value of the object in ***various forms***.

NOTE: ***Type wrappers*** is discussed again in ***autoboxing***.

❒ ***Scanner class:*** Another way to convert a ***numeric string*** into its ***internal, binary format*** is to use one of the methods defined by the ***Scanner*** class, packaged in ***java.util***. **Scanner** reads **formatted** (that is, human-readable) input and converts it into its **binary** form.

☛ **Scanner** can be used to read input from a variety of sources, including the **console** and **files** (**Keyboard** too).

☛ To use **Scanner** to read from the **keyboard**, you must first create a **Scanner** linked to ***console input*** using the *constructor*.

Scanner(InputStream from)

- This creates a ***Scanner*** that uses the stream specified by ***from*** as a **source for input**. You can use this constructor to create a ***Scanner linked to console input***, as : ***Scanner conin = new Scanner(System.in);***
  - ✓ This works because ***System.in*** is an object of type ***InputStream***. After this line executes, ***conin*** can be used to read input from the ***keyboard***.

☛ Once you have created a ***Scanner***, it is a simple matter to use it to read ***numeric input***. Here is the general procedure:

[1] Determine if a specific type of input is available by calling one of ***Scanner's hasNextX*** methods, ***X*** is desired data type.

[2] If input is available, read it by calling one of ***Scanner's nextX*** methods.

☛ Scanner defines two sets of methods that enable you to read input.

- The first are the ***hasNext*** methods. These include methods such as ***hasNextInt()*** and ***hasNextDouble()***. Each of the ***hasNext*** methods returns ***true*** if the desired ***data type*** is the next available item in the data ***stream***, and ***false*** otherwise. Eg: calling ***hasNextInt()*** returns ***true*** only if the next item in the stream is ***text integer***.
- If the desired data is available, you can read it by calling one of Scanner's ***next*** methods, such as ***nextInt()*** or ***nextDouble()***. These methods convert the human-readable form of the data into its internal, binary representation and return the result. Eg: to read an **integer**, call ***nextInt()***.

This shows how to read an **integer** from the **keyboard**. Using this code, if you enter the number ***123*** on the keyboard, then ***i*** will contain the value ***123***.

```
Scanner conin = new Scanner(System.in);
int i;
if(conin.hasNextInt()) i = conin.nextInt();
```

Technically, you can call a ***next method*** without first calling a ***hasNext*** method. However, doing so is not usually a good idea. If a ***next method*** cannot find the type of data it is looking for, it throws an ***InputMismatchException***. For this reason, it is best to first confirm that the desired type of data is available by calling a ***hasNext*** method before calling its corresponding ***next method***.

Example: following demonstrates ***parseInt()*** and ***parseDouble()***. It averages a list of numbers entered by the user.

☛ It first asks the user for the number of values to be averaged. It then reads that number using ***readLine()*** and uses ***parseInt()*** to convert the ***string*** into an ***integer***.

☛ Next, it inputs the values, using ***parseDouble()*** to convert the ***strings*** into their ***double*** equivalents.

```
import java.io.*;
class AvgNums {  public static void main(String args[]) throws IOException {
        BufferedReader br =  new BufferedReader(new InputStreamReader(System.in));
        String str;
        int n;
        double sum = 0.0;
        double avg, t;
        System.out.print("How many numbers will you enter: ");

        str = br.readLine();
        try { n = Integer.parseInt(str); }          // Convert string to int
        catch(NumberFormatException exc) {          System.out.println("Invalid format");
                                                    n = 0;     }
        System.out.println("Enter " + n + " values.");
        for(int i=0; i < n ; i++) {     System.out.print(": ");
                                        str = br.readLine();
                                        try {t = Double.parseDouble(str);   /* Convert string to double */}
                                        catch(NumberFormatException exc) {
                                                    System.out.println("Invalid format");
                                                    t = 0.0; }
                                        sum += t;     }
        avg = sum / n;
        System.out.println("Average is " + avg);   }}
```

SAMPLE RUN:

```
How many numbers will you enter: 5
Enter 5 values.
: 1.1
: 2.2
: 3.3
: 4.4
: 5.5
Average is 3.3
```