Chapter 9

A Few Exceptions, a Little Input/Output, and Some Persistent Objects

From Computing Fundamentals with Java,
Rick Mercer
Franklin, Beedle and Associates

Summing Up

So far, programs have obtained their input from the keyboard. All output has been to the console or a graphical component. All objects have been kept alive only while the program was running. Any changes made to objects were not recorded for later use. In addition, when exceptional events such as `NullPointerException` and `IndexOutOfBoundsException` occurred, they were not handled. Instead, the program terminated.

Coming Up

This chapter introduces a few details about handling exceptions, along with a few of Java's input and output classes. This is a short treatment of both topics. Instead of attempting to explain all exception handling and Java's 60 input and output classes, this chapter will present just enough to show how objects can be made to persist between program runs. The topics are introduced because they are interesting unto themselves. These topics also allow the bank teller system to be completed according to the specifications of Chapter 3.

Using exception handling and the `java.io` package, objects will be written to a file on a disk. The next time the program runs, the objects can be read from that same file. In this way, objects can be saved between program runs. Such objects are said to be persistent. The first section of this chapter introduces Java's exception handling capabilities. This is necessary since using objects from `java.io` requires you to think about what to do when a file is not found or the input was not the expected type. After studying this chapter, you will be able to

- handle a few exceptional events
- use some of Java's 60+ input/output classes
- save the current state of any Java object to a file for later use
- read in objects from files
- see how objects persist in the final phase of the bank teller case study

9.1 A Few Exceptions and How to Handle Them

When programs run, errors occur. Perhaps the user enters a string that is supposed to be a number. When it gets parsed with `parseInt` or `parseDouble`, the method discovers it's a bad number. Or perhaps an arithmetic expression results in division by zero. Or an array subscript is out of bounds. Or there is attempt to read a file from a floppy disk, but there is no disk in the floppy disk drive. Or perhaps a file with a specific name simply does not exist. Exceptional events that occur while a program is running are known as **exceptions**. Programmers have at least two options for dealing with these types of errors.

1. Ignore the exceptional event and let the program terminate
2. Handle the exceptional event

Java specifies a large number of exceptions that may occur while programs run. When Java's runtime system recognizes that some exceptional event has occurred, it constructs an `Exception` object containing information about the type of exception. The runtime system then **throws** an exception. Java's runtime system then searches for code that can handle the exception. If no exception handling code is found, the program terminates.

Consider the example of when a user enters a `String` that does not represent a valid number. During the `parseDouble` message, the code recognizes this exceptional event and throws a `NumberFormatException`.

```
public class HandleException {
  public static void main(String[] args)  {
    Scanner keyboard = new Scanner(System.in);
    System.out.print("Enter a number: ");
    String numberAsString = keyboard.next ();
    double number = Double.parseDouble(numberAsString);

    System.out.println(numberAsString + " stored in number as " + number);
  }
}
```

Dialogue (when the number is valid)

```
Enter a number: 123
123 stored in number as 123.0
```

Dialogue (when the number is NOT valid)

```
Enter a number: 1o1
Exception in thread "main" java.lang.NumberFormatException: 1o1
   at java.lang.FloatingDecimal.readJavaFormatString(FloatingDecimal.java:1176)
   at java.lang.Double.parseDouble(Double.java:184)
   at HandleException.main(HandleException.java:10)
```

The second dialog shows that an exception was thrown when `main` called the `parseDouble` method, which in turn called the `readJavaFormatString` method. These three methods are a stack of methods, with `main` at the bottom of the stack of method calls. The `readJavaFormatString` method is at the top of the stack—where the exception occurred (`main` called `parseDouble` called `readJavaFormatString`). The second dialogue also shows that the program terminated prematurely. The `println` statement at the end of the `main` method never

executed.

It is impossible to predict when a user will enter an invalid number. But the chances are very good that it will happen. One choice is to let the program terminate. The other choice is to handle the exception is some appropriate manner and let the program continue. Exceptions can be handled by writing code that "catches" the exception.

Java allows you to *try* to execute methods that may throw an exception. The code exists in a **`try`** block—the keyword `try` followed by the code wrapped in a block, `{ }`.

```
try {
   code that may throw an exception when an exceptional events occurs
}
catch (Exception anException) {
    code that executes when an exception is thrown
}
```

A `try` block must be followed by a `catch` block—the keyword `catch` followed by the anticipated exception as a parameter and code wrapped in a block. The `catch` block contains the code that executes when the code in the `try` block results in an exception. Because all exception classes extend the `Exception` class, the type of exception could always be `Exception`. In this case, the `catch` block catches any type of exception that can be thrown. However, it is recommended that you use a specific exception that is likely to be thrown by the code in the `try` block, such as `NumberFormatException`, `IndexOutOfBoundsException`, or `IOException`.

The following `main` method provides an example of handling a `NumberFormatException`. This exception handling code (in the `catch` block) executes only when the code in the `try` block throws an exception. This avoids premature program termination when the input string contains an invalid number.

```
public class HandleException {
  public static void main(String[] args) {
    Scanner keyboard = new Scanner(System.in);

    System.out.print("Enter a number: ");
    String numberAsString = keyboard.next();
    double number;

    try {
     // The parseDouble method states that it may throw a NumberFormatException
     number = Double.parseDouble(numberAsString);
    }
    catch(NumberFormatException nfe){
     // Execute this code whenever parseDouble throws an exception
     System.out.println(numberAsString + " is not a valid number");
     System.out.println("Setting number to -1.0");
     number = -1.0;
    }
    System.out.println(numberAsString + " stored in number as " + number);
  }
}
```

Dialogue (when the exception is handled)

```
Enter a number: 1o1
1o1 is not a valid number
Setting number to -1.0
1o1 stored in number as -1.0
```

Instead of ignoring the possibility of exceptional events at runtime, this program now handles potential exceptions by setting the number to an arbitrary value of `-1.0`.

To successfully handle exceptions, a programmer must know if a method might throw an exception, and if so, the type of exception. This is done through documentation. For example, here is the `parseDouble` method which states that the method may throw a `NumberFormatException` and the reason why.

```
/** From the Double class
  *
  * Return a floating-point number represented by the String argument.
  * If numberAsString does not represent a valid number, this method
  * will throw a number format exception.
  */
  public static double parseDouble(String numberAsString)
                                                  throws NumberFormatException
```

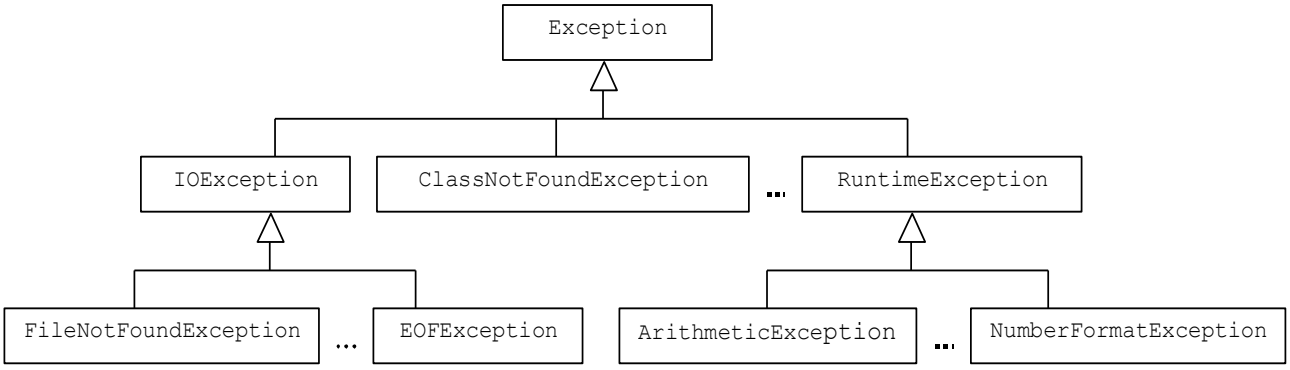
The `parseDouble` method does not catch the exception. Instead, `parseDouble` specifies that it will throw the exception if the exceptional event occurs. A programmer may put a call to this particular method into a `try` block that in turn requires a `catch` block. Then again, the programmer may call this method without placing it in a `try` block. The option comes from the fact that `NumberFormatException` extends `RuntimeException`. A `RuntimeException` need not be handled. Exceptions that don't need to be handled are called **unchecked exceptions**, (`NumberFormatException` is an unchecked exception). The unchecked exception classes are those that extend `RuntimeException`, plus any `Exception` that you write that also extends `RuntimeException`. The unchecked exceptions include the following types (this is not a complete list):

1. `ArithmeticException`
2. `ClassCastException`
3. `IllegalArgumentException`
4. `IndexOutOfBoundsException`
5. `NullPointerException`

Other types of exceptions require that the programmer handle them. These are called **checked exceptions**. Examples of these will be shown later in this chapter with objects from the `java.io` package.

Runtime Exceptions

Java has many different types of exceptions. They are organized into a hierarchy of `Exception` classes. Here are just a few. (*Note:* Those that extend `RuntimeException` need not be handled, but may be handled.)



`RuntimeException`s can be ignored at your own risk. Code that may cause a `RuntimeException` need not be placed in a `try` block, but it can be. Here are some situations that may result in a `RuntimeException` (some code examples are given below).

1. A call to `parseDouble` (or `parseInt`) when the `String` argument does not represent a valid number (see the example above).
2. An integer expression that results in division by zero.
3. Sending a message to an object when the reference variable has the value of `null`.
4. An indexing exception, such as attempting to access an `ArrayList` element with an index that is out of range.

The compiler does not check `RuntimeExceptions`. This means that you do not have to use the `try` and `catch` blocks. If an exceptional event occurs in the following program examples, each program will terminate.

1. Example of integer division by 0 (*Note:* 7.0/ 0.0 returns the value `Infinity`)

```
int numerator = 5;
int denominator = 0;
int quotient = numerator / denominator;
```

Output when an `ArithmeticException` is thrown by Java's runtime system

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
   at OtherRunTimeExceptions.main(OtherRunTimeExceptions.java:7)
```

2. Example of a `null` pointer exception—sending a message to a reference variable that is `null`

```
String str = null;
String strAsUpperCase = str.toUpperCase();
```

Output when a `NullPointerException` is thrown by Java's runtime system

```
Exception in thread "main" java.lang.NullPointerException
   at OtherRunTimeExceptions.main(OtherRunTimeExceptions.java:6)
```

3. Example of an indexing exception

```
List<String> stringList = new ArrayList<String>();
stringList.add("first");
stringList.add("second");
String third = stringList.get(2);   // 0 and 1 are okay
```

Output when an `IndexOutOfBoundsException` is thrown by `ArrayList`'s `get` method

```
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index: 2, Size: 2
    at java.util.ArrayList.RangeCheck(ArrayList.java:491)
    at java.util.ArrayList.get(ArrayList.java:307)
    at OtherRunTimeExceptions.main(OtherRunTimeExceptions.java:10)
```

Runtime exceptions can occur in many places. Even though the compiler does not require that you catch (handle) `RuntimeException`s, as shown above, you can catch them if you want.

```
// Create a list of size 2 (only 0 and 1 can be used as indexes here)
ArrayList<String> stringList = new ArrayList<String>();
stringList.add("first");
stringList.add("second");
Scanner keyboard = new Scanner(System.in);

String choiceFromList;
try {
  // There is no element at index 2. The following message causes an exception.
  System.out.print("Which element [range is 0.." + stringList.size() + "]? ");
  int index = keyboard.nextInt();
  choiceFromList = stringList.get(index);
}
catch(IndexOutOfBoundsException iobe)
{
  System.out.println("index was not in the range of 0.." +
                     (stringList.size() - 1));
  System.out.println("Setting choiceFromList to '??'");
  choiceFromList = "??";
}
System.out.println("choiceFromList was set to " + choiceFromList);
```

Output when an `IndexOutOfBoundsException` is thrown and handled in the `catch` block

```
Which element [range is 0..2]? 2
index was not in the range of 0..1
Setting choiceFromList to '??'
choiceFromList was set to ??
```

Self-Check

9-1 Which of the following statements throws an exception?

a.
```
int j = 7 / 0;
```

b.
```
double x = 7.0 / 0.0;
```

c.
```
String[] names = new String[5];
names[0] = "Austen";
System.out.println(names[1].toUpperCase());
```

9-2 The `ArrayList get` message throws an `IndexOutOfBounds` exception when there is no element at the index passed as an argument. (The index 0 is used here on an empty list.)

```
import java.util.ArrayList;
public class HandleIt {
  public static void main(String[] args) {
    java.util.ArrayList<String> list = new java.util.ArrayList<String>();
    System.out.println(list.get(0));
  }
}
```

Output

```
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index:0, Size:0
        at java.util.ArrayList.RangeCheck(ArrayList.java:491)
        at java.util.ArrayList.get(ArrayList.java:307)
        at HandleIt.main(HandleIt.java:7)
```

Rewrite the code in `main` so that when `get` finds that the index is out of bounds, the `IndexOutOfBounds` exception is caught and "Index out of range" is output. Here is the documentation for the `get` method from Java API:

```
/** From the ArrayList class
  *
  * Returns the element located at position specified by index. This method
  * will throw an exception if index < 0 or index >= this.size().
  */
  public E get(int index) throws IndexOutOfBoundsException
```

Other Methods That Throw Exceptions

Many Java methods are declared to throw an exception. Any method you write can also throw an exception. Consider a method that attempts to enforce this precondition: "The `BankAccount deposit` method requires a deposit amount greater than 0.0." If the amount is less than or equal to 0.0, the method may throw an `IllegalArgumentException`. A message is used as the argument to `IllegalArgumentException` to help further explain the exceptional behavior. (*Note:* The `throws` clause is not required, but it provides good documentation.)

```
// Precondition: depositAmount > 0.0
public void deposit(double depositAmount) throws IllegalArgumentException {
  if(depositAmount <= 0.0) {
    // Create additional help for the programmer to determine the cause of the error
    String errorMessage
        = "\nDeposit amount for '" + this + "' was <= 0.0: " + depositAmount;

    // Construct a new exception that may be caught
    RuntimeException anException
        = new IllegalArgumentException(errorMessage);

    // Throw an exception that could be ignored (the compiler will not check this)
    throw anException;
  }

  // This won't execute with a deposit amount <= 0.0
  my_balance = my_balance + depositAmount;
}
```

Now a message with an argument that violates the precondition results in an exception.

```
BankAccount anAccount = new BankAccount("Jo", 500.00);
anAccount.deposit(-1.00);
```

Output (when program terminates)

```
Exception in thread "main" java.lang.IllegalArgumentException:
Deposit amount for 'Jo $500.00' was <= 0.0: -1.0
        at BankAccount.deposit(BankAccount.java:35)
        at BankAccount.main(BankAccount.java:116)
```

The Java keyword `throw` must be followed by an instance of a class that extends `Throwable`.

Because `Exception` extends `Throwable`, an instance of any `Exception` class in Java can be thrown. In addition, all `Exception` classes in Java have two constructors:

1. a constructor that takes one `String` argument (as shown above), and
2. a default constructor with no arguments (as shown below).

Here is another version of `deposit` that uses the default `Exception` constructor and combines the code into one statement. (*Note:* Since the `IllegalArgumentException` class is in `java.lang`, no `import` is necessary.)

```
public void deposit(double depositAmount) throws IllegalArgumentException {
  if(depositAmount <= 0.0)
    throw new IllegalArgumentException();

  my_balance = my_balance + depositAmount;
}
```

Now when an "illegal" argument is passed to `deposit`, the output is simpler.

Output

```
Exception in thread "main" java.lang.IllegalArgumentException
        at BankAccount.deposit(BankAccount.java:35)
        at BankAccount.main(BankAccount.java:116)
```

In general, throwing an `IllegalArgumentException` object is a reasonable way to handle situations when an argument cannot be used correctly by the method. This can help you and other programmers while debugging.

Self-Check

9-3 Write the `tryIt` method to throw an `IllegalArgumentException` whenever its `Object` parameter is `null`. When the `Object` argument is not `null`, print the `toString` version of the argument. The following code should generate output similar to that shown (assume that you are writing `tryIt` as a method in the `TryAnException` class).

```
TryAnException tae = new TryAnException();
tae.tryIt(new Object());
tae.tryIt(null);
```

Output

```
java.lang.Object@111f71
Exception in thread "main" java.lang.IllegalArgumentException
        at TryAnException.tryIt(TryAnException.java:7)
```

9.2 Input/Output Streams

Most applications require the movement of information from one place to another. In Java, input is read through input stream objects. Output is written through output stream objects. A **stream** is a sequence of items read from some input source or written to some output destination. The input source could be the keyboard, a network connection, or a file on a disk in your computer. The output destination could be the computer screen, the speakers, or a disk file.

These stream objects come from classes in the `java.io` package. This standard library of

many classes allows programmers to define streams that suit a variety of purposes. Some streams allow "raw" bytes to be read or written so pictures and sound can be moved from a source to a destination. Other classes provide functionality such as reading an entire line of input as a `String` (`BufferedReader`) and writing any of Java's objects (`ObjectOutputStream` and `PrintStream`). Typically, two existing Java classes are required to perform input and output. With some knowledge of the Java's stream classes and some work up front, programmers can derive virtually any input and output stream with the desired functionality. Let's first consider how Java handles standard input from the keyboard and standard output to the screen.

Standard Input and Output

Java provides three stream objects for handling keyboard input and screen output. Here are two, the second of which you have already been using:

1. `System.in` The "standard" input stream object (an instance of `InputStream`)
2. `System.out` The "standard" output stream object (an instance of `PrintStream`)

By default, `System.out` writes output to the window where the program is running. The object named `out` is part of the `System` class, which has utilities such as returning the computer system's time (`currentTimeMillis`) and providing standard input and output capabilities. The reference variable `out` is an instance of the `PrintStream` class.

```
/** An object in the System class
  *
  * out is an object already constructed to provide print and println
  * messages of any Java object or primitive. This output stream
  * will commonly display output to the computer's console (screen).
  */
  public static final PrintStream out
```

The `PrintStream` object named `out` has methods to display all of Java's primitive types with `print` and `println` messages. It also has a `print` and `println` method for displaying any Java object. Here are a few of the method headings required to be part the `PrintStream` class.

```
public void print(double d)    // Print a floating-point number
public void print(int i)       // Print an integer
public void print(Object obj)  // Print an object
public void println(boolean x) // Print a boolean and then terminate the line
public void println(char x)    // Print a character and then terminate the line
public void println(char[] x)  // Print an array of characters
public void println(double x)  // Print a double and then terminate the line
public void println(Object x)  // Print an object and then terminate the line
```

The message `System.out.println(1.2)` prints a double, while `System.out.println(5)` prints an integer. `System.out` makes generating output convenient. However, there are no equivalent methods for reading numbers. The input object named `System.in` only reads bytes.

The code to input a number or `String` from the keyboard requires knowledge of Java exception handling. Obtaining input with standard Java classes requires more work and more details. That is the reason why the author-supplied `TextReader` class is included with this textbook.

Although the `InputStream` object only has methods for reading bytes and arrays of bytes, it plays a role in reading numbers and `String`s from the keyboard. The bytes read by `System.in`

must be translated into the appropriate type of data. To get closer to reading doubles and strings, `System.in` is used as an argument to construct an `InputStreamReader` object.

```
InputStreamReader bytesToChar = new InputStreamReader(System.in);
```

The `bytesToChar` object can now read characters from the keyboard with `InputStreamReader`'s `read` method. When an `InputStreamReader` object is constructed with the argument `System.in`, keyboard input can stream into the program at the rate of one character per `read` message.

```
/** From the InputStreamReader class
  *
  * Read one character from the input source. A read message that
  * encounters the end of file will return -1. This method returns
  * the integer equivalent of the character. This means you must store
  * the character into an int, and then perhaps cast the int to a char.
  */
  public int read() throws IOException
```

Notice that this `read` method declares that it may throw an `IOException`. This `IOException` is checked (it does not extend `RuntimeException`). The following attempt to read a character from the keyboard results in a compiletime error:

```
InputStreamReader bytesToChar = new InputStreamReader(System.in);
int anInt = bytesToChar.read();
```

Compiletime Error

```
unreported exception java.io.IOException; must be caught or declared to be thrown
    int anInt = bytesToChar.read();
                            ^
```

There are two ways to handle this checked exception. The simple way is to circumvent Java's exception handling. The method with the `read` message can be declared to throw an `IOException`.

```
// A method declaring that it might throw an IOException gets past the compiler.
import java.io.InputStreamReader;
import java.io.IOException;
public class DeclareExceptionToBeThrown {
                                          // Circumvent exception handling
  public static void main(String[] args) throws IOException  {
    InputStreamReader bytesToChar = new InputStreamReader(System.in);

    System.out.print("Enter a character: ");
    int anInt = bytesToChar.read();

    // To see the character rather than its integer equivalent, cast int to char
    System.out.println((char)anInt);
  }
}
```

Dialogue

```
Enter a character: G
G
```

The second option is to read the message inside of a `try` block. The associated `catch` block will catch the `IOException` should it occur (which is very unlikely when reading from the

keyboard).

The following program reads three characters from the keyboard and prints them. Since `read` returns an `int` instead of a `char`, the return values must be cast from `int` to `char`.

```
// Read three characters using an InputStreamReader.
// This time, the read messages are inside a try block since
// the possible IOException is a checked exception.
import java.io.InputStreamReader;
import java.io.IOException;

public class ReadThreeCharacters {
  public static void main(String[] args) {
    InputStreamReader bytesToChar = new InputStreamReader(System.in);
    int intOne = 0;
    int intTwo = 0;
    int intThree = 0;

    System.out.println("Enter three characters");
    try {
      intOne = bytesToChar.read();
      intTwo = bytesToChar.read();
      intThree = bytesToChar.read();
    }
    catch(IOException ioe) {
      System.out.println("Could not read keyboard input: " + ioe);
    }

    System.out.print((char)intOne);
    System.out.print((char)intTwo);
    System.out.println((char)intThree);
  }
}
```

Output (*Note:* `read` treats blank spaces as valid `char`s, so the ends of lines may be treated as two characters)

```
Enter three characters
a_b
a_b
```

The code shown so far only reads single characters. However, programs are usually more interested in reading strings and numbers. To accomplish this, the `InputStreamReader` object now becomes an argument in the construction of a `BufferedReader` object. The `BufferedReader` class has a method named `readLine` that gets us closer to reading the desired data (with the help of `parseDouble` and `parseInt`).

```
/** From the BufferedReader class
  *
  * Read an entire line of text, which is all character up to
  * but not including the new line char ('\n'). A readLine message
  * that encounters the end of file will return null.
  */
  public String readLine() throws IOException
```

With the following construction, the object `keyboard`—an instance of `BufferedReader`—will understand the `readLine` message.

```
BufferedReader keyboard = new BufferedReader(bytesToChar);
```

The following program can read entire lines of input returned as a `String` (as long as `readLine` is in a `try` block or the `main` method is declared with `throws IOException`).

```
// Read an entire line from the keyboard with BufferedReader's readLine method.
import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.io.IOException;

public class ReadOneLineAsAString {
  public static void main(String[] args)   {
    InputStreamReader bytesToChar = new InputStreamReader(System.in);
    BufferedReader keyboard = new BufferedReader(bytesToChar);
    String line = "";

    System.out.println("Enter a line of text");
    try {
      line = keyboard.nextLine();
    }
    catch(IOException ioe) {
      System.out.println("Could not read keyboard input: " + ioe);
    }

    System.out.println("You entered this line: ");
    System.out.print(line);
  }
}
```

Dialog

```
Enter a line of text
The 3-year-old fox jumped over the 12-year-old dog.
You entered this line:
The 3-year-old fox jumped over the 12-year-old dog.
```

This is as close as Java stream objects get to directly reading integers and doubles. Assuming the line of input contains no blank spaces, the `String` returned by `readLine` can now be passed to `Double`'s `parseDouble` or `Integer`'s `parseInt` method to translate the `String` into a number. If the `String` does not represent a valid number, `parseInt` or `parseDouble` may throw a `NumberFormatException`. Since this is not handled by `try`/`catch`, this program may still terminate prematurely.

```
// Read an integer and a double with the help of several classes and methods
import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.io.IOException;

public class ReadTwoNumbers {
  public static void main(String[] args) {
    InputStreamReader bytesToChar = new InputStreamReader(System.in);
    BufferedReader keyboard = new BufferedReader(bytesToChar);
    String intAsString = "";
    String doubleAsString = "";

    try {
      System.out.print("Enter an int: ");
      intAsString = keyboard.nextLine();
      System.out.print("Enter a double: ");
      doubleAsString = keyboard.nextLine();
    }
```

```
      catch(IOException ioe)  {
        System.out.println("Could not read keyboard input: " + ioe);
      }

      int theInt = Integer.parseInt(intAsString);
      double theDouble = Double.parseDouble(doubleAsString);
      System.out.println("int: " + theInt);
      System.out.println("double: " + theDouble);
    }
}
```

Dialog

```
Enter an int: 123
Enter a double: 4.56
int: 123
double: 4.56
```

The approach just shown for reading numbers and strings from the keyboard is much more complex for the programmer than using a class like `TextReader`. However, the Java stream classes will work on any platform that has Java installed. You can do input without having to copy `TextReader` to your machine. In addition, this approach for numeric input represents the consistent manner in which Java I/O streams read and write data from files, even over the Internet.

In summary, you first construct an object to read bytes from the keyboard, a file, or the Internet. This object is then used to construct another object with easier-to-use methods. The two objects work together to convert raw bytes into something more useable—a `String`.

Although it may be unnecessary to use `try` to read from the keyboard, a stable source of input, most input streams are not nearly as reliable. A server may be down, a network connection may fail, the supplied uniform resource locater (URL) may not exist, or the file name may be misspelled.

Now consider how Java uses the same approach to read input from a file. It is about the same as reading from the keyboard. Input was read from a file earlier with the `TextReader` class. Now you will see how Java handles this with classes that come with Java.

Self-Check

9-4 Complete the following `main` method so it reads one number and prints that number squared. If the user enters an invalid number, display a message indicating so. This means you will need to handle the `NumberFormatException` also.

```
// Read an integer and a double with the help of several classes and methods
import java.io.*;

public class ReadOneInteger {
  public static void main(String[] args) {
    InputStreamReader bytesToChar = new InputStreamReader(System.in);
    BufferedReader keyboard = new BufferedReader(bytesToChar);
    String doubleAsString = "";

    // Write the exception handling code here
  }
}
```

Dialogue 1

```
Enter a number: 4.2
4.2 squared =  17.64
```

Dialogue 2

```
Enter a number: NG
NG was an invalid number
```

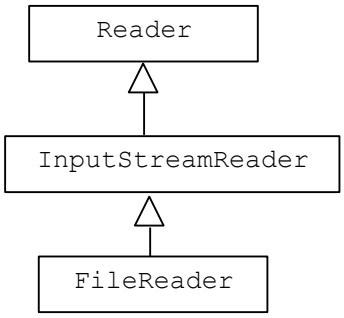
File Input of a Textual Nature

The previous section showed how to read numeric data from the keyboard. It used `BufferedReader` for its handy `readLine` method and `InputStreamReader` to converts bytes typed at the keyboard into characters. This section shows how to get input data from a file. An instance of `BufferedReader` will still be used for its `readLine` method. But this time, an instance of `FileReader` will be used to convert the raw bytes of a file into characters.

To read from the keyboard, the `BufferedReader` object is constructed with an instance of `InputStreamReader`. To read from a file, a `BufferedReader` object can be constructed with an instance of `FileReader`. Both types of arguments are valid because the constructor for `BufferedReader` has a parameter of type `Reader`. In Java, both `FileReader` and `InputStreamReader` are classes that extend the `Reader` class. Both can be passed as arguments to this constructor for `BufferedReader`:

```
/** A method from the Buffereader class
  *
  * Construct an object that uses in to read characters so the
  * BufferedReader object can provide the easy to use readLine method.
  */
  public BufferedReader(Reader in)
```

Both an `InputStreamReader` object, such as `System.in`, and a `FileReader` object can be used to construct a `BufferedReader` object with its `readLine` method. To illustrate, here is a small part of Java's `io` class hierarchy that shows that an instance of `InputStreamReader` can be assigned up the inheritance hierarchy to `Reader`.



Since `FileReader extends InputStreamReader`, an instance of `FileReader` can also be assigned up to the `Reader` parameter in the `BufferedReader` constructor. Remember, assignment up the inheritance hierarchy is valid.

FileReader

To read from a file, you first construct a `FileReader` object with a `String` argument. The `String` represents the name of the file containing the desired information. If the file is in the working folder (directory), only the file name needs to be supplied. The path is not always necessary.

Java forces us to consider the possibility that a file may not exist with that name. A `FileReader` is a checked exception (it does not extend `RuntimeException`). In this case, the `Exception` that could be thrown is `FileNotFoundException`. The following code constructs an object that can read characters from the file named `2numbers.data` (the very small input file is shown below).

```
String fileName = "2numbers.data";
try {
   bytesToChar = new FileReader(fileName);
}
catch(FileNotFoundException fnfe) {
  System.out.println("Could not find file: " + fnfe);
}
```

File `2numbers.data`

```
123
4.56
```

Now that there is the `FileReader` object `bytesToChar`, a `BufferedReader` object can be constructed that has a `readLine` method that is easier to use.

```
BufferedReader inputFile = new BufferedReader(bytesToChar);
```

Multiple `catch` Blocks

The code needed to read from a file could end up throwing three different types of exceptions:

1. file not found
2. input error when attempting to read a line from a file
3. parse errors when converting a `String` to an integer or a double

To make it easier to handle all three, the code for the first two types is placed in one `try` block that is followed by two `catch` blocks. The first `catch` block is for when the file is not found. The second `catch` block is necessary for the exception that may be thrown in `readLine`. The third `catch` block will handle number-format exceptions.

```
// Input numbers from a file on a disk
import java.io.FileReader; // Reading from a file this time
import java.io.BufferedReader;
import java.io.IOException;
import java.io.FileNotFoundException; // File may not be found

public class ReadTwoNumbersFromADiskFile {
  public static void main(String[] args) {
    // These variables are needed in more than one block. Therefore they must
    // be declared here so they are accessible to all blocks in this main method.
    String fileName = "2numbers.data";
    String intAsString = "";
    String doubleAsString = "";
    int theInt = 0;
    double theDouble = 0.0;

    try {
      // Constructing a new FileReader needs to be tried.
      // It may throw FileNotFoundException.
      FileReader bytesToChar = new FileReader(fileName);
```

```
        // If the FileReader is okay, this should work.
        BufferedReader inputFile = new BufferedReader(bytesToChar);

        // Read an integer from the first line and a double from line two.
        // Both of these readLine messages need to be tried.
        // They may throw an IOException.
        intAsString = inputFile.readLine();
        doubleAsString = inputFile.readLine();

        // Now done with the input file.
        inputFile.close();

        // Now try to parse the input strings (could still get an exception here).
        theInt = Integer.parseInt(intAsString);
        theDouble = Double.parseDouble(doubleAsString);
      }
      catch(FileNotFoundException fnfe) {
        System.out.println("Could not find file: " + fnfe);
      }
      catch(IOException ioe) {
        System.out.println("Could not read from file named: " + fileName);
      }
      catch(NumberFormatException nfe) {
        System.out.println("One of the numbers was bad: " + fileName);
      }

      // This output always occurs
      System.out.println("int: " + theInt);
      System.out.println("double: " + theDouble);
    }
  }
```

Output (to console window when input is from a file and everything goes okay)

```
int: 123
double: 4.56
```

This is what happens when the file is not found.

Output (when the file does not exist)

```
Could not find file: java.io.FileNotFoundException: 2numbers.data
(The system can not find the file specified)
Terminating program
```

Although the program terminates prematurely here, the exception handling code needs to be changed. For example, the `catch` block could to do something more useful, such as ask the user for another file name (it could use a loop).

Even if the file is found and the two lines of input exist in the file, exceptions can still occur. Here is another exception that is thrown by `parseInt` when just one seemingly innocent blank space trails the text `123`. The `parseInt` method tries to convert the string `"123 "`.

Output (when there is a trailing blank after `123`—the string is `"123 "`)

```
Exception in thread "main" java.lang.NumberFormatException: 123
        at java.lang.Integer.parseInt(Integer.java:414)
        at java.lang.Integer.parseInt(Integer.java:454)
        at ReadTwoNumbersFromADiskFile.main(ReadTwoNumbersFromADiskFile.java:45)
```

This particular error could be fixed with `String`'s `trim` method that strips leading and trailing blanks from the `String`.

```
int theInt = Integer.parseInt(intAsString.trim());
```

Text file input requires a program to be aware of exactly how the input data exists in the file. It is all too easy to have an unexpected value, such as a `double` instead of an `int`, a trailing space, or a blank line at the end of the file.

FileWriter

Java's `FileWriter` class provides the ability to *write* characters to a file. Like `FileReader`, `FileWriter` provides low-level disk access capabilities. A `FileWriter` object is constructed with a `String` argument representing the name of the file being written to. However, the attempt to construct one could throw an `IOException`. The file name could be malformed, for example. So constructing a `FileWriter` object must be handled with `try`/`catch`.

```
String outputFileName = "output.data";

FileWriter charToBytesWriter = null;
try {
  charToBytesWriter = new FileWriter(outputFileName);
}
catch(IOException ioe) {
  System.out.println("Could not create the new file: " + ioe);
}
```

At this point, `charToBytesWriter` can write characters and `String`s to a file. To obtain the familiar `print` and `println` methods that write all objects and primitive types, construct a `PrintWriter` object (`PrintWriter` is very similar to `System.out`). The following code constructs a `PrintWriter` object so any Java value can be `print`(ed) or `println`(ed).

```
// PrintWriter's constructors do not throw any exceptions
PrintWriter diskFile = new PrintWriter(charToBytesWriter);

// Now diskFile can understand print and println
diskFile.print("First line has one number: ");
diskFile.println(123);
diskFile.println(4.56);
diskFile.println('c');
diskFile.println(3 < 4);
diskFile.print(diskFile);
```

Ensuring that the Data Is Actually Written

Writing data to a disk is slow. To make programs run faster, Java places the output data first into a buffer (perhaps an array of bytes). Then at some point, a large number of bytes are written to the disk at once. Writing a large number of bytes a few times is much faster than writing one byte a large number of times. The consequence of implementing this more efficient way of writing to a disk is that the program may terminate before the data is actually written to the disk! To ensure that all data is written to the disk, close the output stream with a `close` message. Without the following `close` message, the output file would be empty—nothing would actually be written.

```
// Must explicitly close the file or risk losing data that
// is buffered--waiting to actually be written.
diskFile.close();
```

The `close` message to an output stream ensures that the output buffer has actually been written to the disk. This newly created file has six different Java values written to it and stored on a computer disk.

Output (written to the file named `output.data`)

```
First line has one number: 123
4.56
c
true
java.io.PrintWriter@720eeb
```

Destroying Files

If a new `FileWriter` object is constructed with a file name that already exists, the old file is destroyed. While this may often be precisely what you want, there are times when this is not desirable. Java has a class named `File` that allows you to manage files and directories. For example, you could construct a `File` object and ask it to search for an existing file. For now, just be aware that constructing a `FileWriter` object deletes any existing file with the same name.

The `java.io` package has a large set of classes for performing input and output of characters, strings, pictures, and sound from a variety of sources and to a variety of destinations. Complete coverage of these 60+ classes in `java.io` is beyond the scope of this book. However, a few more are presented next. In particular, the next section will use objects from `java.io` that can make objects persist from one program run to the next.

Self-Check

9-5 `FileWriter` has a method for writing a `String` to a disk. Using this Java documentation:

`public void write(String str)` *In class FileWriter*
`throws IOException`

Write a string.
Parameters: str - String to be written
Throws: IOException - If an I/O error occurs

complete this `main` method so that it writes your name to a file named `myname.txt`.

```
import java.io.*;
public class WriteNameToDisk {

  public static void main(String[] args)  {
     String outputFileName = "myname.txt";
     FileWriter charToBytesWriter = null;

     // Complete the code to write your name here

  }
}
```

9.3 Object Streams for Persistent Objects

If data is not stored on a disk or some other media, it will disappear when the program terminates. The program must do something extra to save data that is needed later. Using the few streams just

described, objects could be saved to a disk by taking them apart, instance variable by instance variable, and writing the pieces to the disk. Objects could be restored by reading in the pieces from the disk and constructing them again. But this is a lot of unnecessary work. The code gets even more complex when the objects have arrays and other complex structures. Java brings a much more elegant solution for making objects live between program runs. The Java tools for doing this are Java's object streams and object serialization.

A **persistent object** is one that can be saved for later use. To make objects persist, the class and all instance variables must be `Serializable`. Java arrays, many Java classes, and all primitives are `Serializable`. However, to make any new object `Serializable`, the class must state that it implements the `Serializable` interface.

```
public class BankAccount implements Serializable {
  private String my_ID;
  private double my_balance;
  // ...
}
```

There are no methods in the `Serializable` interface. The `Serializable` tag only identifies the class as being `Serializable`. In addition to tagging the class as `Serializable`, you must ensure that all instance variables are `Serializable`. Since `double` and `String` are already `Serializable`, no further action would be required for the `BankAccount` class shown above.

Assuming that a class and its instance variables are serializable, the next step is to write serialized objects to a destination such as a file. To make persistent objects, you can use Java's `ObjectOutputStream` class with its `writeObject` method for this. The constructor for `ObjectOutputStream` needs a `FileOutputStream` object for writing the bytes to a disk. Once again, two streams work together to get the desired outcome—a stream that saves any serialized object to a file.

```
FileOutputStream bytesToDisk = new FileOutputStream("fileName");
ObjectOutputStream objectToBytes = new ObjectOutputStream(bytesToDisk);
```

The following program declares that the `main` method may throw an `IOException`. This allows the code in the method to get past the compiler that is watching for checked exceptions.

```
// Make one object persist for another program to read later.
import java.io.*;

public class WriteAnObjectToDisk {

  // This method does not have any exception handling code.
  // The program will terminate if something goes wrong.
  public static void main(String[] args) {
    // Construct one object from a class not supplied by Java.
    // To write this BankAccount object to a disk, the class and all of
    // its instance variables must implement Serializable (this is the case).
    BankAccount singleAccount = new BankAccount("ID123", 100.00);
    singleAccount.withdraw(50.00);
    singleAccount.deposit(12.75);

    try {
      // Build the stream that can write objects (not text) to a disk
      FileOutputStream bytesToDisk
            = new FileOutputStream("BankAccount.object");
      ObjectOutputStream objectToBytes
            = new ObjectOutputStream(bytesToDisk);
```

```
        // Show the state of the account before saving it
        System.out.println("About to write this object to disk: ");
        System.out.println(singleAccount);

        // Now objectToBytes will understand the writeObject message.
        // Make the object persist on disk, so it can be read later.
        objectToBytes.writeObject(singleAccount);

        // Do NOT forget to close the output stream
        objectToBytes.close();
      }
      catch(IOException ioe) {
        System.out.println (ioe.toString());
      }

    }
}
```

Output

```
About to write this object to disk:
ID123 $62.75
```

Now that this `BankAccount` object has been saved in a file, another program may read this object. The program must know the name of the file where the object has been written. It must also know the class of object that is stored in that file.

```
// Read an object stored in a file
import java.io.*;

public class ReadAnObjectFromDisk {
  // This method may throw an IOException when the file is not found.
  public static void main(String[] args) throws IOException  {
    // First create an object input stream with the readObject method
    FileInputStream diskToStreamOfBytes
        = new FileInputStream("BankAccount.object");

    // Construct an objectNow with the readObject method
    ObjectInputStream objectToBytes
        = new ObjectInputStream(diskToStreamOfBytes);

    // Read the entire object with the ObjectInputStream. The checked exception
    // must be caught (even though Object is a known class).
    Object anyObject = null;
    try {
      anyObject = objectToBytes.readObject();
    }
    catch(ClassNotFoundException cnfe) {
      System.out.println(cnfe);
    }

    // Now cast from Object to the class that it is known to be: BankAccount.
    BankAccount singleAccount = (BankAccount)anyObject;

    // Close input files also.
    objectToBytes.close();

    // And show that the state is the same as the other program that wrote it.
    System.out.println("The object just after reading it from disk");
    System.out.println(singleAccount);
  }
}
```

Output (this `BankAccount` has the same state as that shown above)

```
The object just after reading it from disk
ID123 $62.75
```

Other Persistent Objects

Many existing Java classes implement `Serializable`. This means a large number of objects can be made to persist this way. Programs can read objects from a disk in the same manner as was shown with the `BankAccount`. The objects can be quite complex. Java provides the mechanisms for storing any object and then later retrieving it in the same state that it was written.

Consider the following program that shows seven different persistent objects. This program simplifies the `try`/`catch` blocks by placing all checked exceptions into one `try` block. If any exception is thrown—and there are several classes of exceptions that could be thrown—the correct `catch` block prints the exception information.

To save lists of different objects, the following classes must implement `Serializable` (the Java classes already do and `BankAccount` has already been made `Serializable`).

1. `ArrayList`
2. `BankAccount`
3. `Integer`
4. `Double`
5. `String`
6. `Character`
7. `Date`

```
// Write seven classes of objects to a disk, including ArrayList
import java.io.*;    // FileOutputStream and ObjectOutputStream
import java.util.*;  // ArrayList and Date

public class WriteManyObjectsToDisk {
  public static void main(String[] args) {
    ArrayList<Object> polyList = new ArrayList<Object>();
    polyList.add(new BankAccount("ID123", 100.00)) ;
    polyList.add(new Integer (123));
    polyList.add(new Double (4.56));
    polyList.add(new String("A String"));
    polyList.add(new Character('G'));
    polyList.add(new Date());   // Today's date

    try  {
      FileOutputStream bytesToDisk
          = new FileOutputStream("ArrayList.object");
      ObjectOutputStream objectToBytes
          = new ObjectOutputStream(bytesToDisk);

      // Show the state before saving it
      System.out.println("About to write this object to disk: ");
      System.out.println(polyList);

      // Now objectToBytes will understand the writeObject message.
      // Make the object persist on a disk, so it can be read later.
      objectToBytes.writeObject(polyList);

      // Do NOT forget to close the output stream
      objectToBytes.close();
    }
```

```
        catch(FileNotFoundException e) {
          System.out.println("File not found " + e);
        }
        catch(IOException e) {
          System.out.println("Error during ObjectOutputStream construction,\n" +
                             "or writing object, or file close. " + e);
        }
      }
    }
```

Output

```
About to write this object to disk:
[ID123 $100.00, 123, 4.56, A String, G, Mon Oct 08 17:31:02 MST 2002]
```

If any one of these objects were not `Serializable`, an exception would be thrown during the `writeObject` message. Consider adding a new `AnotherClass` object that is implemented like this:

```
public class AnotherClass extends Object {
}
```

An instance of `AnotherClass` could always be added to the `ArrayList`.

```
polyList.add(new AnotherClass());
```

Once this message adds the `AnotherClass` object at the end of the `ArrayList`, the program results in the following output.

Output (when a new `AnotherClass` is added at the end of the `ArrayList`)

```
About to write this object to disk:
[ID123 $100.00, 123, 4.56, A String, G, Sat Jun 23 20:13:53 MST 2001, AnotherClass@1a698a]
```

However, the `writeObject` message would then throw a `NotSerializableException`, as evidenced by this output where the exception was caught.

Output (an error is reported while trying to write an object that is not serialized)

```
java.io.NotSerializableException: AnotherClass
```

The write will fail. It will not save the other objects either. To save all different objects in this `ArrayList`, change the `AnotherClass` class so it is serializable:

```
class AnotherClass implements Serializable {
}
```

There is not much state to store for `AnotherClass`, but if instance variables were added to `AnotherClass`, then those objects must also be `Serializable`. The next section shows other examples of objects that are written to and read from a disk. The bank teller system will be completed.

Key Terms

`catch`	`NumberFormatException`	stream
exception	`ObjectInputStream`	`throw`
`FileNotFoundException`	`ObjectOutputStream`	`try`
`FileReader`	persistent object	`windowClosing`
`FileWriter`	`readObject`	`WindowListener`
`IOException`	`RuntimeException`	`writeObject`
`java.io`	`showConfirmDialog`	

Chapter Summary

- When programs run, errors occur.
- Java has unchecked exceptions that extend `RuntimeException`. Examples include integer division by 0 and array and string indexes that are out of range. Code that may throw these exceptions may be placed in a `try` block.
- Java has checked exceptions that force programmers to consider that some exceptional event may occur. Examples include trying to read data from a file and trying open a file that does not exist. Code that may throw these exceptions must be in a `try` block.
- Exceptions can be handled with `try` and `catch` to avoid programs terminating on their own.
- Java has a large number of classes that allow programmers to create input and output streams in a large variety of ways, including input from the keyboard.
- Most Java streams use two classes, one for low-level reading or writing of bytes, such as `InputStream` and `FileReader`. Another class supplies higher-level methods, such as `readLine` in `BufferedReader` and `readObject` in `ObjectInputStream`.
- Persistent objects are those that live after a program terminates.
- Complex objects can be read in and stored in their entirety. The class and its instance variables must be `Serializable`.
- The `WindowListener` interface requires seven methods for listening to window events such as `windowClosing`. `JFrame` objects generate window events.

Programming Tips

1. You have not seen all there is to exception handling.

Exception handling was only introduced in this chapter. You may want to refer to advanced books to see some related topics that are beyond the scope of this textbook, including

- other exception classes in Java
- nested `try`/`catch` blocks
- the `finally` block

2. You have not seen all the input and output streams.

Java streams were only introduced in this chapter. You may want to refer to advanced books to see some related topics that are beyond the scope of this textbook, such as the `File` class, random access files, and reading numbers from compressed files. However, one more example of reading input (over the Internet) is described as a programming project at the end of this chapter.

3. You can circumvent exception handling in Java.

When you are dealing with marked exceptions, the code that throws them must be enclosed in a `try` block, which in turn requires a `catch` block. You are forced to handle the exception, even if you do nothing in the `catch` block. In this way your program continues to run even if the exception is thrown. You can also simply claim that your method throws the exception. Then, when the exception is thrown, it is handled by some other method. You have no control over what happens.

4. You usually need two streams for input or output.

It is common to have two or more input streams to read input. It is common to have two or more output streams to write output. The Java I/O classes were designed to work together to build a wide variety of input streams and output streams. This provides great flexibility. For example, an input stream could come in over the Internet. The downside is that you need to write some extra code.

5. Always close output streams.

If you are writing to an output stream, the output may be buffered. This means the writing physically transfers data after a buffer—an array perhaps—is filled. You may have some data that is written to the buffer, but not physically transferred to its final destination. The `close` message ensures that all data is copied from the buffer to the destination. If you don't close the stream, you will most assuredly lose some data.

6. All instance variables must be serialized to write and read objects.

An object you want to write to a disk must be serialized. Also, you must make sure all of the instance variables in each object are serialized. For example, it is not enough to serialize `BankAccountCollection`. The `BankAccount` class and anything it contains must also be `Serializable`. Tagging the classes with `implements Serializable` does this.

Exercises

1. Given the following data, for each of the following, which exception will be thrown?

```
int number = 5;
String oneName = "Harry";
String[] names = new String[number];
```

a. `System.out.println(8 / (number - 5));`
b. `names[number] = oneName;`
c. `System.out.println(oneName.substring(3, number + 1));`

2. What exceptions are unchecked? What does it mean when you need to write code that may throw an unchecked exception?

3. The `get` message in the following code throws an `IndexOutOfBounds` exception when the user enters a negative index or `index >= list.size()`. Two dialogues show the different behavior.

```
ArrayList<String> list = new ArrayList<String>();
list.add("first");
list.add("second");
```

```
Scanner keyboard = new Scanner(System.in);
if(list.isEmpty())
   System.out.print("The list is empty");
else {
  // Get index in range to find the value of the element in the list
  System.out.print("Enter an index from 0.." + (list.size() - 1) + ": ");
  int index = keyboard.nextInt();
  System.out.println("The String at " + index + ": " + list.get(index));
}
System.out.println("That's all folks");
```

Dialogues (the second execution throws an `IndexOutOfBoundsException`)

```
Enter an index from 0..1: 1
The String at 1: second
That's all folks

Enter an index from 0..1: 2
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index: 2, Size: 2
        at java.util.ArrayList.RangeCheck(ArrayList.java:491)
        at java.util.ArrayList.get(ArrayList.java:307)
        at A.main(A.java:18)
```

Rewrite the code in the `else` block so it prints "Index out of range, try again" until the user enters an index that is in the range.

```
Enter an index from 0..1: -2
Index out of range, try again
Enter an index from 0..1: 2
Index out of range, try again
Enter an index from 0..1: 1
The String at 1: second
That's all folks
```

4. Write a minimal program that reads keyboard input and saves it in the file selected by the user. Use stream classes from the `java.io` package. Assume all imports have been included and you are writing the code in a method that `throws IOException`. One dialogue should look like this:

```
Save keyboard input into which file? exercise.txt

Enter lines or 'SENTINEL' as the last line:
This is keyboard input
using standard java classes.
As each line is entered, it is saved to a disk file
until there is one line with SENTINEL
SENTINEL
```

5. Write the minimal code to add to the previous exercise that prints all of the lines from the file that was saved to disk (including `SENTINEL`).

6. Write the minimal code to construct one `BankAccount` object with user input and then save it as a serialized object. Use stream classes from the `java.io` package. Assume all imports have been included and you are writing the code in a method that `throws IOException`. One dialogue should look like this:

```
Enter account name: Kay
Enter initial balance: 105
About to write this object to disk:
Kay $105.00
```

7. Write the minimal code to add to the previous exercise that reads the serialized `BankAccount` object and prints it.

```
The object after reading the persistent object:
Kay S $105.00
```

Answers to Self-Check Questions

9-1

a. Throws `DivisionByZeroException`

b. Does not throw an exception

c. Throws `NullPointerException`

9-2

```
try {
  System.out.println("Object at index 0: " + list.get(0));
}
catch(IndexOutOfBoundsException iobe) {
  System.out.println("Index out of bounds");
}
```

9-3

```
// precondition: obj is not null
public void tryIt(Object obj) throws IllegalArgumentException {
  if(obj == null)
    throw new IllegalArgumentException();
  else
    System.out.println(obj.toString());
}
```

9-4

```
try {
  System.out.print("Enter a number: ");
  doubleAsString = keyboard.nextLine();
}
catch(IOException ioe) {
  System.out.println("Could not read keyboard input: " + ioe);
}

try {
  double theDouble = Double.parseDouble(doubleAsString);
  System.out.println(theDouble + " squared =  " + theDouble * theDouble);
}
catch (NumberFormatException nfe) {
  System.out.println(doubleAsString + " was an invalid number");
}
```

9-5

```
try {
  charToBytesWriter = new FileWriter(outputFileName);
  charToBytesWriter.write("Your Name");  // <-- Must be in a try block
  charToBytesWriter.close();
}
catch(IOException ioe)
{
  System.out.println("Could not create the new file: " + ioe);
}
```