Chapter 11, Files and Exceptions

John M. Morrison

February 13, 2021

Contents

0 Introduction **2**

1 Exceptions **2**

2 Throwable **4**

3 Throwing **6**

4 Checked v. Run-Time **7**

5 Path **8**

6 Reading a Text File **11**

7 Buffering **15**

8 Writing a Text File **18**

9 Binary/Buffered **20**

0 Introduction

This chapter will introduce you interacting with the file system. At the same time, it will be necessary to learn about the concept of exception handling. The reason for this is that when you access a file and it is not present or you do not have permission to read it, an exception is generated.

These exceptions are beyond programmer and, often user, control. What you don't want them doing is crashing your program. So you will learn how to run code with these sorts of hazards and learn how to recover from the problem at hand gracefully. We will begin this chapter by discussing exceptions.

1 Exceptions

We have already seen these. For example, when creating `BigFraction`, we threw an `IllegalArgumentException` when a client programmer attempted to create a fraction with a zero denominator. There is little doubt you have encountered a `NullPointerException` when you forgot to use `new` to allocate memory for an object or a `IndexOutOfBounds` exception when working with a collection. Consider this unfortunate code.

```
import java.util.ArrayList;
public class DumbError
{
    private static ArrayList<String> roster;

    public static void main(String[] args)
    {
        roster.add("Doofus McDuff");
        System.out.println(roster);
    }
}
```

This code compiles.

```
$ javac DumbError.java
```

Now let us run it.

```
$ java DumbError
Exception in thread "main" java.lang.NullPointerException
    at DumbError.main(DumbError.java:8)
$
```

Doom. You see that a `NullPointerException` got thrown. The compiler did not catch the error; an exception is a run-time error. You will see that on line 8 the offending code is

```
    roster.add("Doofus McDuff");
```

This occured because we never did this

```
roster = new ArrayList<>();
```

We attempted to call a method on an object pointing to the graveyard state `null`. Insert this line we just showed and your program will run without error. Other exceptions you probably have run across include these. All of these exceptions are examples of *run-time exceptions*. Most run-time exceptions are caused by programmer errors. When you encountered them, you saw that your program died on the spot and that a stack trace was generated.

There is another species of exception you are about to meet called *checked exceptions*. These exceptions occur because of events that are beyond user, or programmer, control. These sorts of exceptions crop up when handling files; being able to handle them is prerequisite to doing any interaction with the file system.

Java provides a mechanism called *exception handling* that provides a parallel track of return from functions so that you can avoid cluttering the ordinary execution of code with endless error-handling routines.

Exceptions are objects that are "thrown" by various methods or actions. In this chapter we will learn how to handle (catch) an exception. By so doing we allow our program to recover and continue to work. Failure to catch and exception results in a flood of nasty text from Java (a so-called "exploding heart"). Crashes such as these should be extremely rare in production-quality software. We can use exceptions as a means to protect our program from such dangers as user abuse and from such misfortunes as crashing whilst attempting to gain access to an nonexistent or prohibited resource. Many of these hazards are beyond both user and programmer control.

When you program with files or with socket connections, the handling of exceptions will be mandatory; hence the need for this interlude before we begin handling files.

Let us now turn to understanding how exceptions fit into Java's class hierarchy.

Progrmming Exercise

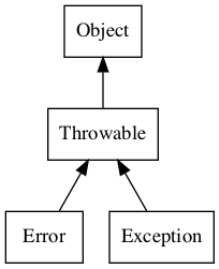
1. What happens if you try to compute 1/0 using integers?
2. What happens if you try to compute 1/0 using doubles?
3. What happens if you make the call `Math.sqrt(-1)`?
4. What happens if you make the call `Math.log(-1)`?

2 The `Throwable` Subtree

Go to the Java API guide and pull up the class `Exception`. The family tree is is as follows.

```
java.lang.Object
java.lang.Throwable
java.lang.Exception
```

and here is an inheritance tree.



The class name `Throwable` is a bit strange; one would initially think it were an interface. It is, however, a class. The class `java.lang.Exception` has a sibling class `java.lang.Error`.

When objects of type `Error` are thrown, it is not reasonable to try to recover. These things come from problems in the Java Virtual Machine, bad memory problems, or problems from the underlying OS. We just accept the fact that they cause program death. Continuing to proceed would just lead to a chain of ever-escalating problems.

Objects of type `Exception` are thrown for more minor problems, such as an attempt to open a non-existent file for reading, trying to convert an unparseable string to an integer, or trying to access an entry of a string, array, or array list that is out of bounds.

Let us show this mechanism at work. For example, if you attempt to execute the code

```
int foo = Integer.parseInt("gobbledegook");
```

you will be rewarded with an exception. To see what happens, create this program `MakeException.java`.

```
public class MakeException
{
```

```
    public static void main(String[] args)
    {
        int foo = Integer.parseInt("gobbledegook");
    }
}
```

This program compiles happily. You will see that the infraction we have here is a run-time error, as is any exception.

When you run the program you will see this acrimonious screed.

```
unix> java MakeException
Exception in thread "main" java.lang.NumberFormatException:
    For input string: "gobbledegook"
at java.base/java.lang.NumberFormatException.forInputString(
       NumberFormatException.java:68)
at java.base/java.lang.Integer.parseInt(Integer.java:652)
at java.base/java.lang.Integer.parseInt(Integer.java:770)
at MakeException.main(MakeException.java:5)
unix>
```

The exploding heart you see here shows a stack trace. This shows how the exception propagates through the various calls the program makes. To learn what you did wrong, you must look in this list for your file(s). You will see the offending line here.

```
    at MakeException.main(MakeException.java:5)
```

You are being told that the scene of the crime is on line 5, smack in the middle of your `main` method. The stack trace can yield valuable clues in tracking down and extirpating a run-time problem such as this one.

We have seen reference to "throw" and "throws" before. Go into the API guide and bring up the class `String`. Now scroll down to the method summary and click on the link for the familiar method `charAt()`. You will see this notation.

Throws:

> `IndexOutOfBoundsException` - if the `index` argument is negative or not less than the length of the string.

Let us now look up this `IndexOutOfBoundsException`. The family tree reveals that this class extends the `RuntimeException` class. The purpose of

this exception is to terminate the execution of the program since you are in an error state. We can, however, run code in response to the occurrence of an exception, so our program does not crash.

3 Throwing an Exception

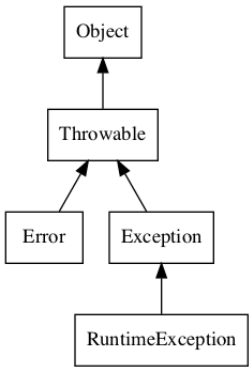
You can throw exceptions when something is about to go wrong. For example in our BigFraction class, we threw an exception if a denominator of zero was passed. Here it is.

```
public BigFraction(BigInteger num, BigInteger denom)
{
    if(denom.equals(BigInteger.ZERO)
    {
        throw new IllegalArgumentException();
    }
    (rest of constructor)
}
```

It is best, if at all possible, to use a standard library exception. You can create your own exceptions by inheriting from any class in the `Exception` subtree. If you inherit from `RuntimeException`, the caller does not have to handle the exception as shown in the next section.

4 Checked and Run-Time Exceptions

There are two types of exceptions that exist: runtime exceptions and all others, which are called checked exceptions. How do you know if an exception is a `RuntimeException`? Just look up its family tree and see if it is a descendant of `RuntimeException`. So far in our study of Java, we have only seen runtime exceptions. Here is the throwable subtree with `RuntimeException` added.



Checked exceptions, on the other hand, are usually caused by situations beyond programmer, or even end-user control. Suppose a user tries to get a program to open a file that does not exist, or a file for which he lacks appropriate permissions. Another similar situation is that of attempting to create a *socket*, or a connection to another computer. The host computer may not allow such connections, it could be down, or it could even be nonexistent. These situations are not necessarily the user's or programmer's fault.

Checked exceptions must be *handled*; this process entails creating code to tell your program what to do in the face of these exceptions being thrown. It is entirely optional to handle a runtime exception.

Sometimes a runtime exception will be caused by user error; in these cases it is appropriate to use exception handling to fix the problem. For example if a user is supposed to enter a number into a `TextField` the hapless fool enters a string that is not numeric, your program might try to use `Integer.parseInt` to convert it into an integer. Here we see a problem created by an end-user. This user should be protected and this error should be handled gracefully so that (bumbling) user can go about his business. You always want to protect the end-user from exceptions if it is at all feasible or reasonable. Remember: *Never reward a paying customer with death.* It's bad for business. Now let us erect some scaffolding for handling files, as we will introduce the idea of handling exceptions in the context of fileIO.

5 The `Path` to Perdition

We begin with the interface `java.nio.files.Path`. It has a list of methods that is specifies for handling locations in your file system. Said locations might or might not exist. Since `Path` is an interface, you cannot create instances of a `Path` using `new`. However, this interface comes equipped with a static method `of` Here is the method detail.

```
static Path of(String first, String...  more)
```

Returns a Path by converting path string, or a sequence of strings that when joined form a path string. If `more` does not specify any elements then the value of the first parameter is the path string to convert. If more specifies one or more elements then each non-empty string, including first, is considered to be a sequence of name elements and is joined to form a path string. The details as to how the Strings are joined is provider specific but typically they will be joined using the name-separator as the separator. For example, if the name separator is `"/"` and `getPath("/foo","bar","gus")` is invoked, then the path string `"/foo/bar/gus"` is converted to a Path. A Path representing an empty path is returned if first is the empty string and more does not contain any non-empty strings.

The Path is obtained by invoking the getPath method of the default FileSystem.

Note that while this method is very convenient, using it will imply an assumed reference to the default FileSystem and limit the utility of the calling code. Hence it should not be used in library code intended for flexible reuse. A more flexible alternative is to use an existing Path instance as an anchor, such as:

```
Path dir = ...
Path path = dir.resolve("file");
```

Parameters:
`first` - the path string or initial part of the path string
`more` - additional strings to be joined to form the path string
Returns:
the resulting Path **Throws:**
`InvalidPathException` - if the path string cannot be converted to a Path
Since:
11
See Also:
FileSystem.getPath(java.lang.String, java.lang.String...)

Note that the class `Paths` merely implement the `of` method of this class. The `Paths` is probably a dead-end class, so you should use `Path.of` in preference to its `get` method. It will be the workhorse for us in this chapter.

Let us now take a tour of `Path` methods. Begin by creating a directory named `zoo` with these contents

```
(base) MAC:Sat Nov 28:10:33:s1> ls -Rl zoo
total 0
-rw-r--r--  1 morrison  staff     0B Nov 27 16:03 capybara
drwxr-xr-x  6 morrison  staff   192B Nov 28 10:32 cats
-rw-r--r--  1 morrison  staff     0B Nov 27 16:03 dingo
-rw-r--r--  1 morrison  staff     0B Nov 27 16:03 eland
drwxr-xr-x  6 morrison  staff   192B Nov 28 10:32 reptiles
```

```
zoo/cats:
total 0
-rw-r--r--  1 morrison  staff     0B Nov 27 16:03 bobcat
-rw-r--r--  1 morrison  staff     0B Nov 28 10:32 cheetah
-rw-r--r--  1 morrison  staff     0B Nov 28 10:32 lion
-rw-r--r--  1 morrison  staff     0B Nov 28 10:32 tiger

zoo/reptiles:
total 0
-rw-r--r--  1 morrison  staff     0B Nov 27 16:03 alligator
-rw-r--r--  1 morrison  staff     0B Nov 28 10:32 anaconda
-rw-r--r--  1 morrison  staff     0B Nov 28 10:32 cobra
-rw-r--r--  1 morrison  staff     0B Nov 28 10:32 iguana
```

Enter this directory and fire up `jshell`. We begin by getting our current working directory as a `Path` and determining its absolute position in our file system. Note the use of `isAbsolute` to tell if a path is absolute or relative.

```
jshell> Path zoo = Path.of(".")
zoo ==> .

jshell> Path absZoo = zoo.toAbsolutePath()
absZoo ==> /Users/morrison/book/S1/zoo/.

jshell> zoo.isAbsolute()
$3 ==> false

jshell> absZoo.isAbsolute()
$4 ==> true
```

Here is a look up the file family tree.

```
jshell> absZoo.getParent()
$8 ==> /Users/morrison/book/S1/zoo

jshell> absZoo.getParent().getParent()
$9 ==> /Users/morrison/book/S1
```

The result is disappointing when we do this on `zoo`, but we sho a hackish workaround here.

```
jshell> zoo.getParent()
$10 ==> null
```

```
jshell> zoo.toAbsolutePath().getParent().getFileName()
$11 ==> zoo
```

OPC Note In older OPC, you will see reference to the class `java.io.File`; this is the predecessor to the modern `Path` interface, which you should prefer. However, you can convert a `Path` object to a `File` by calling the `toFile` method on it. The older `File` class has a `toPath` method to convert from `File` to `Path`. You should prefer the `Path` interface when writing new code.

java.nio.file.Paths This static service class appears to be an abortive effort and you should avoid it. Construct new paths using the static `of` method.

Interacting with the File System The next class for you to know about is `Files`; this is the workhorse class for fileIO. The `Path` interface and `Files` jointly cover all of the territory previously covered by the old `java.io.File` class and a whole lot more.

That name `Files` should give you a hint. You have seen the pluralized class name before in `java.util.Collections` and `java.util.Arrays`.

Alex, give me "common bonds" for $800. Alex replies, "Here is the answer: this is the common bond between these three classes."

The smart contestant says, "What is being a static service class?" Bingo, add $800 to that contestant's score.

To move around in our little file tree and to manipulate it, we will use `Files`. This is a static service class that gives us access to the file system, as well as an array of other useful services..

Boola! Boola! The `Files` class provides some useful predicates to determine file attributes. As you might expect, their names begin with `is`. These four check out a file's type.

- `isDirectory(Path path)` This returns `true` if the path points at a directory.
- `isRegularFile(Path path)` This returns `true` if the path points at a at a regular file.
- `isHidden(Path path)` This returns `true` if the path points at a a hidden (in UNIX a dotfile) file.
- `isSymbolicLink(Path path)` This returns `true` if the path points at a at a symbolic link (an alias for a file or a directory).

These tell you about file permissions.

- `isReadable(Path path)` This returns `true` if the path points at a file you have read permissions for.
- `isWritable(Path path)` This returns `true` if the path points a file you have write permissions for.
- `isExecutable(Path path)` This returns `true` if the path points a file you have execute permissions for.
- `isDirectory(Path path)` This returns `true` if the path points

These two will tell you basic "identity" information.

- `exists(Path path, LinkOption... option )` This returns `true` if the path points at a file that exists. The second argument is entirely optional and we won't bother with it.
- `isSameFile(Path path1, Path path2)` This returns `true` if the both paths point at the same file.

You can also create and delete files and directories.

- `createFile(Path path, FileAttribute<?>... option )` This returns `true` if the path points at a file that exists. The second argument is entirely optional, and allows you to specify file permissions.
- `delete(Path path)` This deletes the file at the specified path. point at the same file.

6 Reading a Text File

Open the API page for the class `java.nio.Files`. We are going to write a class named `Cat.java` which accepts a string as a command-line argument that (should) be a name of an existing file and which puts the file to `stdout`. For the sake of simplicity, we will do this all in the `main` method. Now get the method detail for the static method `readAllLines`. This method reads the lines of the file into a `List<String>`. Let us attempt this.

```
import java.util.List;
import java.nio.file.Files;
import java.nio.file.Path;
public class Cat
{
    public static void main(String[] args)
    {
```

```
        Path path = Path.of(args[0]);
        List<String> fileContents = Files.readAllLines(path);
        for(String s: fileContents)
        {
            System.out.print(s);
        }
    }
}
```

Now we compile and see this.

```
unix> javac Cat.java
Cat.java:9: error: unreported exception IOException;
    must be caught or declared to be thrown
        List<String> fileContents = Files.readAllLines(path);
                                                       ^
1 error
```

Uh oh. Because an `IOException` is a checked exception, action on our part is required. We begin by enclosing our "dangerous code" in a `try` block. This must be followed by a `catch` block for an `IOException`. Note that a new import is needed.

```
import java.util.List;
import java.nio.file.Files;
import java.nio.file.Path;
import java.io.IOException;
public class Cat
{
    public static void main(String[] args)
    {
        try
        {
            Path path = Path.of(args[0]);
            List<String> fileContents = Files.readAllLines(path);
            for(String s: fileContents)
            {
                System.out.print(s);
            }
        }
        catch(IOException ex)
        {
            ex.printStackTrace();
        }
    }
}
```

Now, create a text file to run this on.

```
This is a test
it is only a test.
Let's see if this works.
```

Let's run it.

```
unix> java Cat.java test.txt
This is a testit is only a test.Let's see if this works.unix>
```

Evidently all of the newlines get amputated. Let's put 'em back; just change `System.out.print` to `System.out.println`.

```
unix> java Cat.java test.txt
This is a test
it is only a test.
Let's see if this works.
```

Et Voila!

Can we shorten this code? Yes, if we avail ourselves of the `forEach` method for lists.

```
import java.util.List;
import java.nio.file.Files;
import java.nio.file.Path;
import java.io.IOException;
public class Cat
{
    public static void main(String[] args)
    {
        try
        {
            Path path = Path.of(args[0]);
            List<String> fileContents = Files.readAllLines(path);
            fileContents.forEach(System.out::println);
        }
        catch(IOException ex)
        {
            ex.printStackTrace();
        }
    }
}
```

We are using a *method reference* in the `forEach` method.

Next, for a little perverse fun, let us run our program on a nonexistent file `heffalump.txt`. Fury is unleashed by the JVM.

```
base) MAC:Tue Dec 22:15:57:s9> java Cat.java heffalump.txt
java.nio.file.NoSuchFileException: heffalump.txt
        at java.base/sun.nio.fs.UnixException.translateToIOException(UnixException.java:92)
        at java.base/sun.nio.fs.UnixException.rethrowAsIOException(UnixException.java:106)
        at java.base/sun.nio.fs.UnixException.rethrowAsIOException(UnixException.java:111)
        at java.base/sun.nio.fs.UnixFileSystemProvider.newByteChannel(UnixFileSystemProvider
        at java.base/java.nio.file.Files.newByteChannel(Files.java:375)
        at java.base/java.nio.file.Files.newByteChannel(Files.java:426)
        at java.base/java.nio.file.spi.FileSystemProvider.newInputStream(FileSystemProvider.
        at java.base/java.nio.file.Files.newInputStream(Files.java:160)
        at java.base/java.nio.file.Files.newBufferedReader(Files.java:2916)
        at java.base/java.nio.file.Files.readAllLines(Files.java:3396)
        at java.base/java.nio.file.Files.readAllLines(Files.java:3436)
        at Cat.main(Cat.java:12)
        at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccess
        at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMeth
        at java.base/java.lang.reflect.Method.invoke(Method.java:564)
        at jdk.compiler/com.sun.tools.javac.launcher.Main.execute(Main.java:415)
        at jdk.compiler/com.sun.tools.javac.launcher.Main.run(Main.java:192)
        at jdk.compiler/com.sun.tools.javac.launcher.Main.main(Main.java:132)
```

Scan through this wreckage; it reveals that the source of the exception was on line 12 in our file. Let's handle it and cut down on the Wagnerian drama.

```
import java.util.List;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.NoSuchFileException;
import java.io.IOException;
public class Cat
{
    public static void main(String[] args)
    {
        try
        {
            Path path = Path.of(args[0]);
            List<String> fileContents = Files.readAllLines(path);
            fileContents.forEach(System.out::println);
        }
        catch(NoSuchFileException ex)
        {
            System.err.printf("File %s does not exist.\n", args[0]);
```

```
        }
        catch(IOException ex)
        {
            ex.printStackTrace();
        }
    }
}
```

This is better.

```
unix> java Cat heffalump.txt
File heffalump.txt does not exist.
```

This reveals something else. We can have several `catch` blocks. Order is important. Put the most specific exceptions (lower on the inheritance tree) at the top and the most generals ones at the bottom. Only one `catch` block will ever by executed. Here is one other minor tweak. Dum-dum user just might forget a command-line argument. Just at this at the top of your `main` method before the `try` block.

```
if(args.length == 0)
{
    System.err.println("A command-line argument is required");
}
```

7 Dealing with Dyspepsia

What if we want to process a humongous file? Reading it all at once could be a huge memory hog. Can we be more efficient? Happily the tools are at hand. We will bring `newBufferedReader` to bear on the problem. During this exercise, you will learn about *try with resources* that will automatically close any file you open. Let us rewrite `Cat.java` using this method.

```
import java.util.List;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.NoSuchFileException;
import java.io.BufferedReader;
import java.io.IOException;
public class Cat
{
    public static void main(String[] args)
    {
        if(args.length == 0)
```

```
        {
            System.err.println("A command-line argument is required");
        }
        try
        {
            Path path = Path.of(args[0]);
            BufferedReader br  = Files.newBufferedReader(path);
            String line = "";
            while( (line = br.readLine()) != null)
            {
                System.out.println(line);
            }
            br.close();
        }
        catch(NoSuchFileException ex)
        {
            System.err.printf("File %s does not exist.\n", args[0]);
        }
        catch(IOException ex)
        {
            ex.printStackTrace();
        }
    }
}
```

So what has happened? A `BufferedReader` creates a connection to a file. Hidden from you is the buffer, which stores a chunk of the file in your program's memory. Usually this chunk is of size 4K. In the beginning the `BufferedReader` grabs a chunk of text. We then have it reading the file a line at at time. When the buffer is empty, another chunk of file is hoovered into the buffer. So, if we are reading a large file, our memory footprint is far smaller than if we got the whole file at once using `readAllLines`. Also, we are not pestering the operating system with a zillion requests for chunks of the file.

A problem remains. If an exception occurs, the file might not close. There is a smart way to deal with this called *try with resources*. Let us see what that looks like.

```
import java.util.List;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.NoSuchFileException;
import java.io.BufferedReader;
import java.io.IOException;
public class Cat
{
```

```
    public static void main(String[] args)
    {
        if(args.length == 0)
        {
            System.err.println("A command-line argument is required");
        }
        Path path = Path.of(args[0]);
        try
        (
            BufferedReader br  = Files.newBufferedReader(path);
        )
        {
            String line = "";
            while( (line = br.readLine()) != null)
            {
                System.out.println(line);
            }
        }
        catch(NoSuchFileException ex)
        {
            System.err.printf("File %s does not exist.\n", args[0]);
        }
        catch(IOException ex)
        {
            ex.printStackTrace();
        }
    }
}
```

Notice we moved the declaration of `path` so it is in scope throughout `main`. We no longer need to close the file, because try with resources automatically does this. As a result, a great deal of bother you will never see will never take place.

How do I know if I can use try with resources? Just check to see if the class you are using implements `java.lang.AutoCloseable`. This interface specifies a single method, `public void close()`. Look at the API page; a lot of things implement it. If you write some kind of file-handling class, you should implement it, too.

That `while` loop looks so awful Pssst... Here is a little magic. Ditch this code

```
String line = "";
while( (line = br.readLine()) != null)
```

```
{
    System.out.println(line);
}
```

for this:

```
br.lines()
  .forEach(System.out::println);
```

Ooh, sweet. How does that work? In brief, the call `br.lines()` gives you a Netflix-like streaming view of the file. The lines come in a stream. The `forEach` method applies `System.out.println` to each line in the stream. You have had a preview of the Streams API which will be covered in more detail later. It is a stupendously powerful appratus that will make your code shorter, more readable, and more expressive.

8 Writing a Text File

One might just think that if there is a `BufferedReader` that there could be a `BufferedWriter`. Correct. And now that we know about exception handling, writing to a file should be a fairly simple process. Let's do it. Here an idea. Let's generate an HTML trig table from 0 to 90 degrees for sine and cosine.

What do we need? We will write a function that creates the table rows, and functions that compute sine and cosine in degrees.

```
import java.nio.file.Files;
import java.nio.file.Path;
import java.io.BufferedWriter;
import java.io.IOException;
public class Triggie
{
    private static final double FACTOR = Math.PI/180;
    public static void main(String[] args)
    {
    }
    private static double sinDeg(int x)
    {
        return Math.sin(FACTOR*x);
    }
    private static double cosDeg(int x)
    {
        return Math.cos(FACTOR*x);
    }
```

```
    private static String makeRow(int x)
    {
        return String.format("<tr><td>%d</td><td>%.4f</td><td>%4f</td></tr>\n",
            x, sinDeg(x), cosDeg(x));
    }
}
```

Now we write the `main` method. We will use try with resources.

```
public static void main(String[] args)
{
    Path path = Path.of("trigTable.html");
    try(BufferedWriter bw = Files.newBufferedWriter(path))
    {
        bw.write("<table>\n<tr><th>x</th><th>sin(x)</th><th>cos(x)</th></tr>\n");
        for(int k = 0; k <= 90; k++)
        {
            bw.write(makeRow(k));
        }
        br.write("<table>\n");
    }
    catch(IOException ex)
    {
        ex.printStackTrace();
    }
}
```

Note the pleasing parallelism here.

9 Buffered IO with Binary Files

Here we introduce some new classes in `java.io`. These manage unbuffered and buffered byte streams.

- `java.io.InputStream`
- `java.io.OutputStream`
- `java.io.BufferedInputStream`
- `java.io.BufferedOutputStream`

We will create a program that copies files containing raw bytes. In our example, we will use an image file. This has been tested on a 300 megabyte video file and did the job pleasingly quickly. Let's get started with some basic stuff, inserting the needed imports and wrangling the command-line arguments.

```
import java.nio.file.Files;
import java.nio.file.Path;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.nio.file.NoSuchFileException;
public class BinaryCopy
{
    public static void main(String[] args)
    {
        String donor = args[0];
        String recipient = args[1];
        if(args.length < 2)
        {
            System.err.println("Two command-line arguments are needed");
        }
        Path inFile = Path.of(donor);
        Path outFile = Path.of(recipient);
    }
}
```

Now let us proceed to creating the try with resources header.

```
try
(
    InputStream in = new
        BufferedInputStream(Files.newInputStream(inFile));
    OutputStream out =
        new BufferedOutputStream(Files.newOutputStream(outFile));
)
```

We now append the usual catch blocks.

```
try
(
    InputStream in = new
        BufferedInputStream(Files.newInputStream(inFile));
    OutputStream out =
        new BufferedOutputStream(Files.newOutputStream(outFile));
)
{
}
catch(NoSuchFileException ex)
{
```

```
    System.err.printf("File %s not found.\n", donor);
}
catch(IOException ex)
{
    ex.printStackTrace();
}
```

The coup d'grace is stunningly simple.

```
try
(
    InputStream in = new
        BufferedInputStream(Files.newInputStream(inFile));
    OutputStream out = new
        BufferedOutputStream(Files.newOutputStream(outFile));
)
{
    byte[] bytes = Files.readAllBytes(inFile);
    out.write(bytes);
}
catch(NoSuchFileException ex)
{
    System.err.printf("File %s not found.\n", donor);
}
catch(IOException ex)
{
    ex.printStackTrace();
}
```

The docs recommend against this. However it will copy a file containing several hundred megabytes with pleasing dispatch.