Exceptions

&

File I/O Part 1

Today's Objectives

- Understanding exception handling
- Implementing a try/catch structure in a program
- **`java.io`** library **`File`** and **`Directory`** classes
- Character streams: what are they and how do we use them?
- Using the try-with-resources block
- Handling File I/O exceptions and how to recover from them
- Real world uses for File I/O

Exceptions

What are Exceptions?

Exceptions are occurrences that alter the flow of the program away from the ideal or “happy” path.

- *Sometimes it’s the developer’s fault*: i.e. accessing an array element greater than the actual number of elements present.

- *Other times it’s not*: i.e. loss of internet connection, a data file that was supposed to be there has been removed by a systems admin
.
- Java uses classes inherited from the **`Exception`** class to handle exception situations in code.

Runtime Exceptions

Runtime exceptions are errors that occur while the program is executing in the JVM. Here are three common examples:

- **`NullPointerException`**: A call was made to a method or a data member was accessed for a null reference.
- **`ArithmeticException`**: Code attempted to divide by zero.
- **`ArrayIndexOutOfBoundsException`**: An element of an array was accessed with an index that is out of bounds.

Checked Exceptions

Checked exceptions are exceptions which are defined as needing to be handled or declared in Java code.

- **`FileNotFoundException:`** This is thrown programmatically, when the program tries to do something with a file that doesn’t exist.

- **`IOException:`** A more general exception related to problems reading or writing to a file.

- Note that **`FileNotFoundException`** extends from **`IOException`**.

"Throwing" and Handling Exceptions

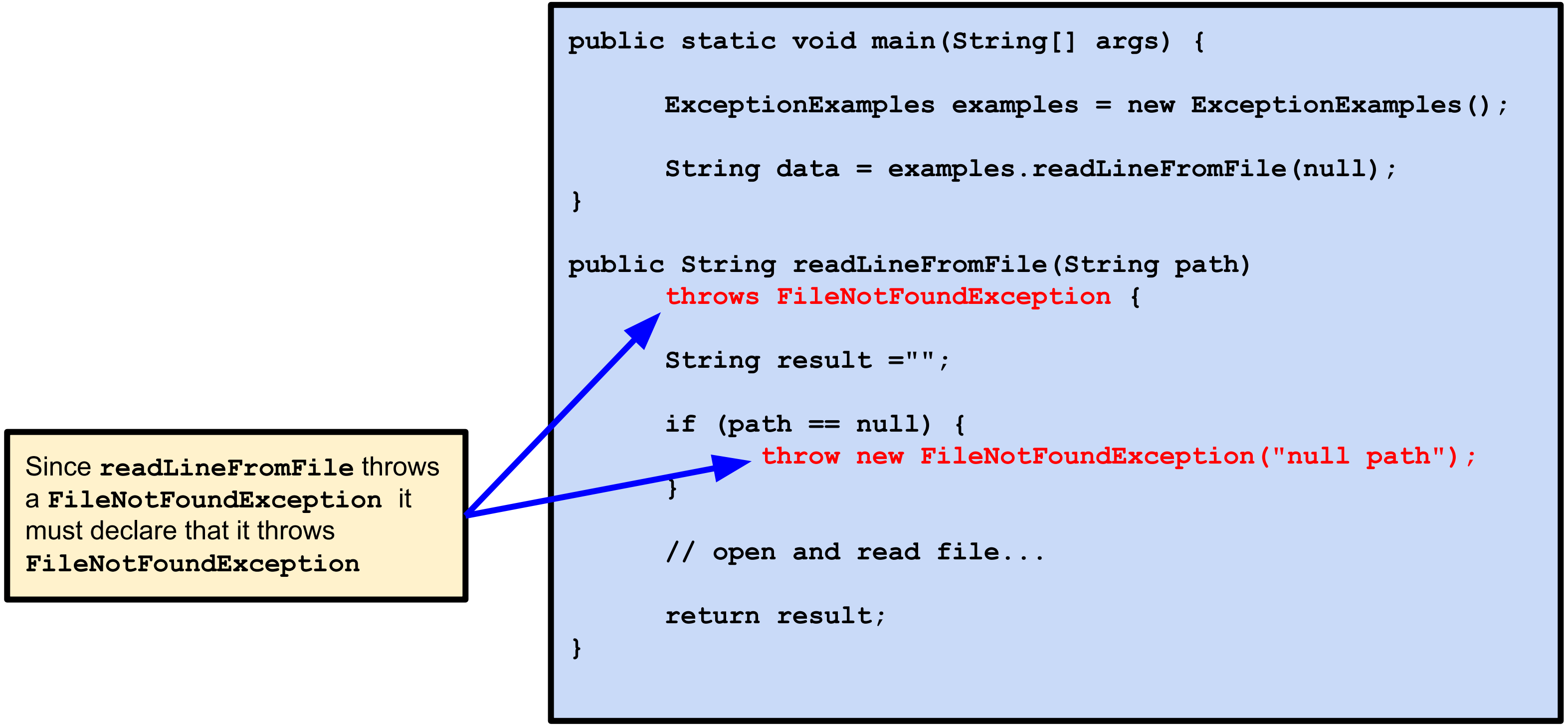
Throwing means making everyone aware that a deviation from normal program flow has occurred. You may also hear this referred to as *raising* an exception but Java uses the "throw" concept.

- The JVM may throw an exception behind the scenes.
- Java code may also explicitly throw exceptions to indicate exception situations have occurred.
- All Exceptions, whether they are Runtime Exceptions or Checked Exceptions, inherit from the **`Exception`** class.

Checked Exception Handling Flow

```
public static void main(String[] args) {

      ExceptionExamples examples = new ExceptionExamples();

      String data = examples.readLineFromFile(null);
}

public String readLineFromFile(String path)
      throws FileNotFoundException {

      String result ="";

      if (path == null) {
            throw new FileNotFoundException("null path");
      }

      // open and read file...

      return result;
}
```

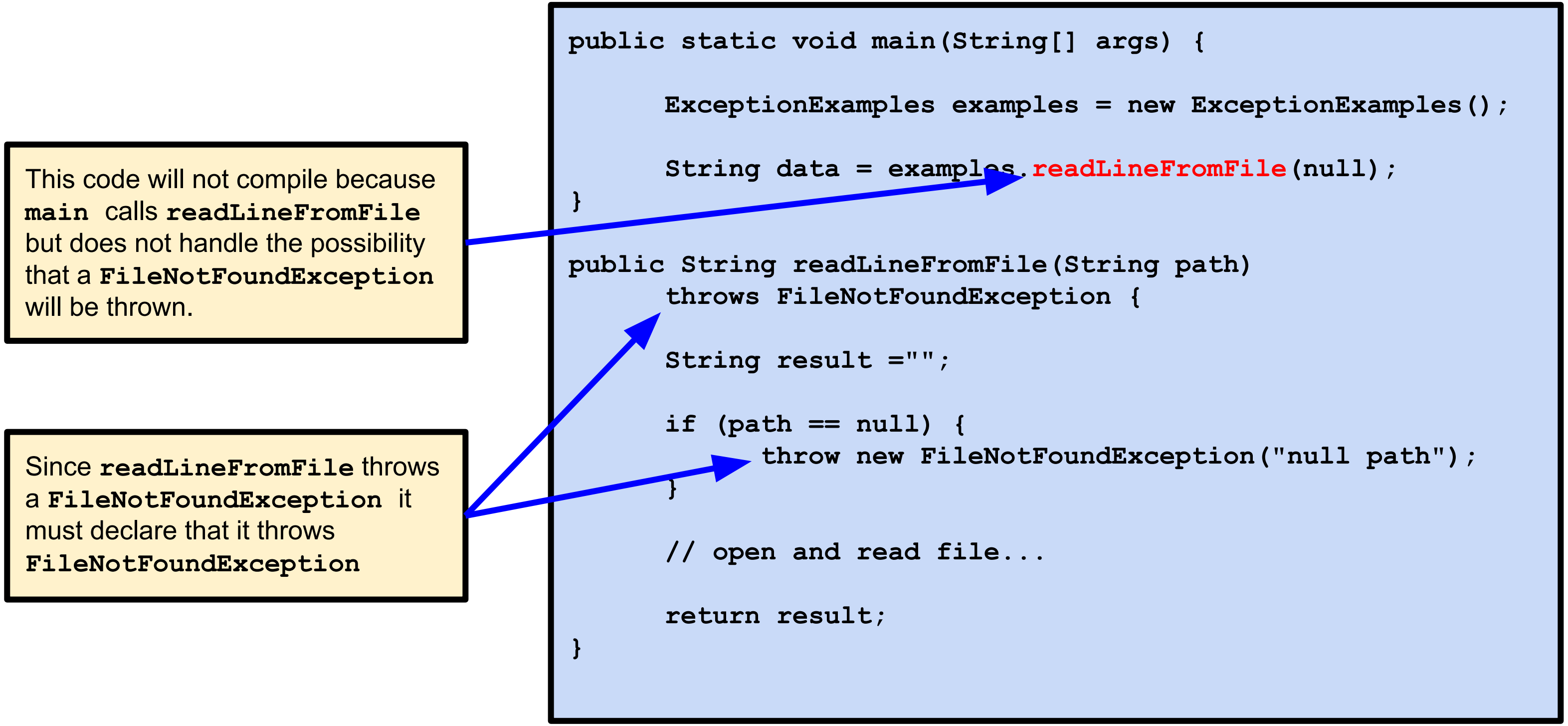
Checked Exception Handling Flow



```
public static void main(String[] args) {

      ExceptionExamples examples = new ExceptionExamples();

      String data = examples.readLineFromFile(null);
}

public String readLineFromFile(String path)
      throws FileNotFoundException {

      String result ="";

      if (path == null) {
            throw new FileNotFoundException("null path");
      }

      // open and read file...

      return result;
}
```

Since **`readLineFromFile`** throws a **`FileNotFoundException`** it must declare that it throws **`FileNotFoundException`**

Checked Exception Handling Flow



This code will not compile because **main** calls **readLineFromFile** but does not handle the possibility that a **FileNotFoundException** will be thrown.

Since **readLineFromFile** throws a **FileNotFoundException** it must declare that it throws **FileNotFoundException**

```
public static void main(String[] args) {

      ExceptionExamples examples = new ExceptionExamples();

      String data = examples.readLineFromFile(null);
}

public String readLineFromFile(String path)
      throws FileNotFoundException {

      String result ="";

      if (path == null) {
            throw new FileNotFoundException("null path");
      }

      // open and read file...

      return result;
}
```

Checked Exception Handling Flow

When a method calls another method which declares that it throws an exception, it has two choices:

- Handle the exception
- Pass the buck up by declaring that it ***also*** throws the specified exception.

```
public static void main(String[] args)
      throws FileNotFoundException {

      ExceptionExamples examples = new ExceptionExamples();

      String data = examples.readLineFromFile(null);
}

public String readLineFromFile(String path)
      throws FileNotFoundException {

      String result ="";

      if (path == null) {
            throw new FileNotFoundException("null path");
      }

      // open and read file...

      return result;
}
```

Checked Exception Handling Flow

When a method calls another method which declares that it throws an exception, it has two choices:

- Handle the exception
- Pass the buck up by declaring that it ***also*** throws the specified exception.

```
public static void main(String[] args)
      throws FileNotFoundException {

      ExceptionExamples examples = new ExceptionExamples();

      String data = examples.readLineFromFile(null);
}

public String readLineFromFile(String path)
      throws FileNotFoundException {

      String result ="";

      if (path == null) {
            throw new FileNotFoundException("null path");
      }

      // open and read file...

      return result;
}
```

We'll look at how to handle the exception shortly, but here **`main`** is passing the buck by declaring that it throws a **`FileNotFoundException`**

Checked Exception Handling Flow

Wrapping code that needs to handle a checked exception in a try-catch block:

- Attempts to execute the code in the try block.
- If it succeeds, it moves past the try-catch block.
- If an exception of the type specified in the catch block is thrown, the code within the catch block will be executed before moving forward.

```
public static void main(String[] args)  {

      ExceptionExamples examples = new ExceptionExamples();

      try {
            String data = examples.readLineFromFile(null);
      } catch (FileNotFoundException e) {
            // do something to handle the exception
      }

}

public String readLineFromFile(String path)
      throws FileNotFoundException {

      String result ="";

      if (path == null) {
            throw new FileNotFoundException("null path");
      }

      // open and read file...

      return result;
}
```

Checked Exception Handling Flow

Wrapping code that needs to handle a checked exception in a try-catch block:

- Attempts to execute the code in the try block.
- If it succeeds, it moves past the try-catch block.
- If an exception of the type specified in the catch block is thrown, the code within the catch block will be executed before moving forward.

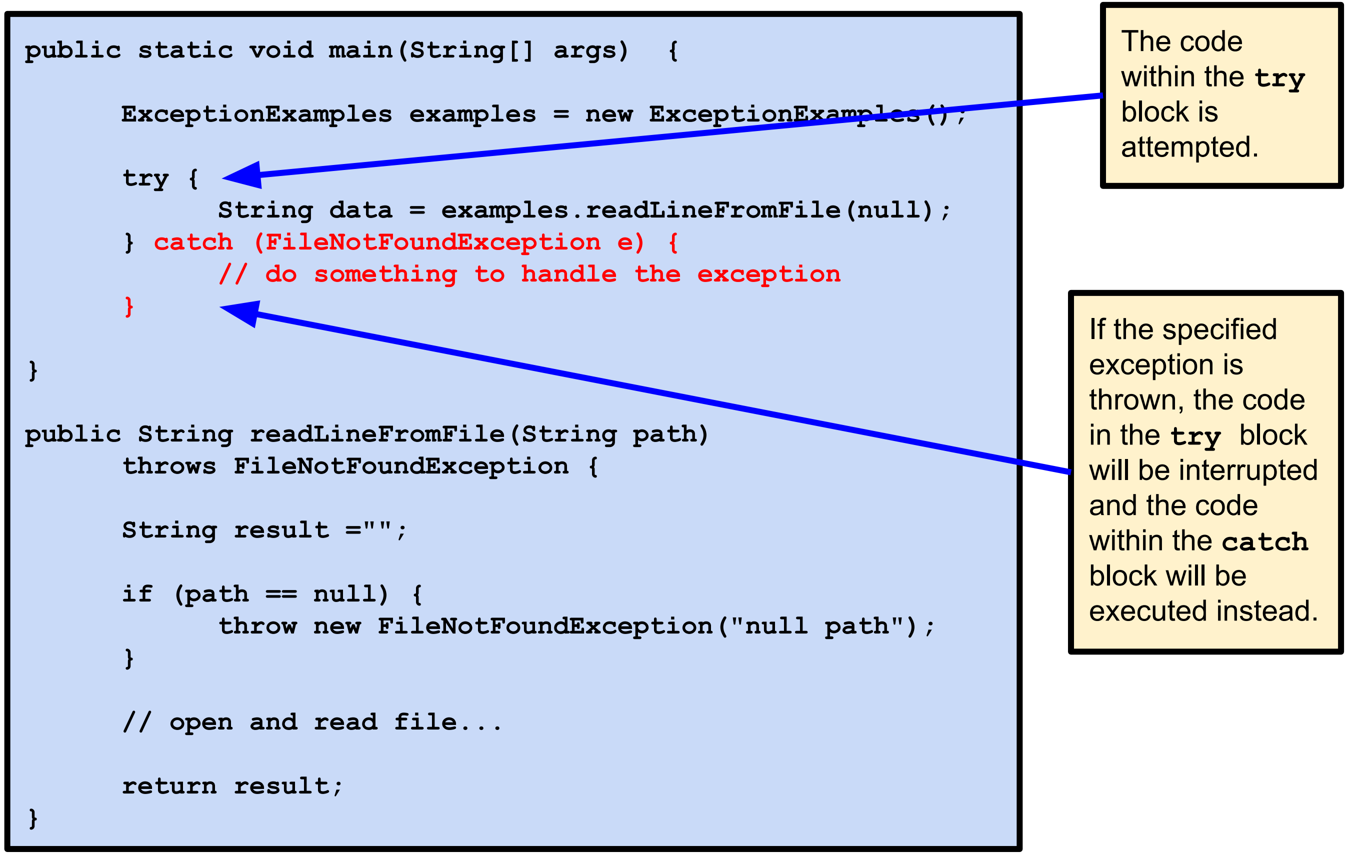
```
public static void main(String[] args)  {

      ExceptionExamples examples = new ExceptionExamples();

      try {
            String data = examples.readLineFromFile(null);
      } catch (FileNotFoundException e) {
            // do something to handle the exception
      }

}

public String readLineFromFile(String path)
      throws FileNotFoundException {

      String result ="";

      if (path == null) {
            throw new FileNotFoundException("null path");
      }

      // open and read file...

      return result;
}
```

The code within the **try** block is attempted.

Checked Exception Handling Flow

Wrapping code that needs to handle a checked exception in a try-catch block:

- Attempts to execute the code in the try block.
- If it succeeds, it moves past the try-catch block.
- If an exception of the type specified in the catch block is thrown, the code within the catch block will be executed before moving forward.



```
public static void main(String[] args)  {

      ExceptionExamples examples = new ExceptionExamples();

      try {
            String data = examples.readLineFromFile(null);
      } catch (FileNotFoundException e) {
            // do something to handle the exception
      }

}

public String readLineFromFile(String path)
      throws FileNotFoundException {

      String result ="";

      if (path == null) {
            throw new FileNotFoundException("null path");
      }

      // open and read file...

      return result;
}
```

The code within the **try** block is attempted.

If the specified exception is thrown, the code in the **try** block will be interrupted and the code within the **catch** block will be executed instead.

A Closer Look at the Try-Catch-Finally Block

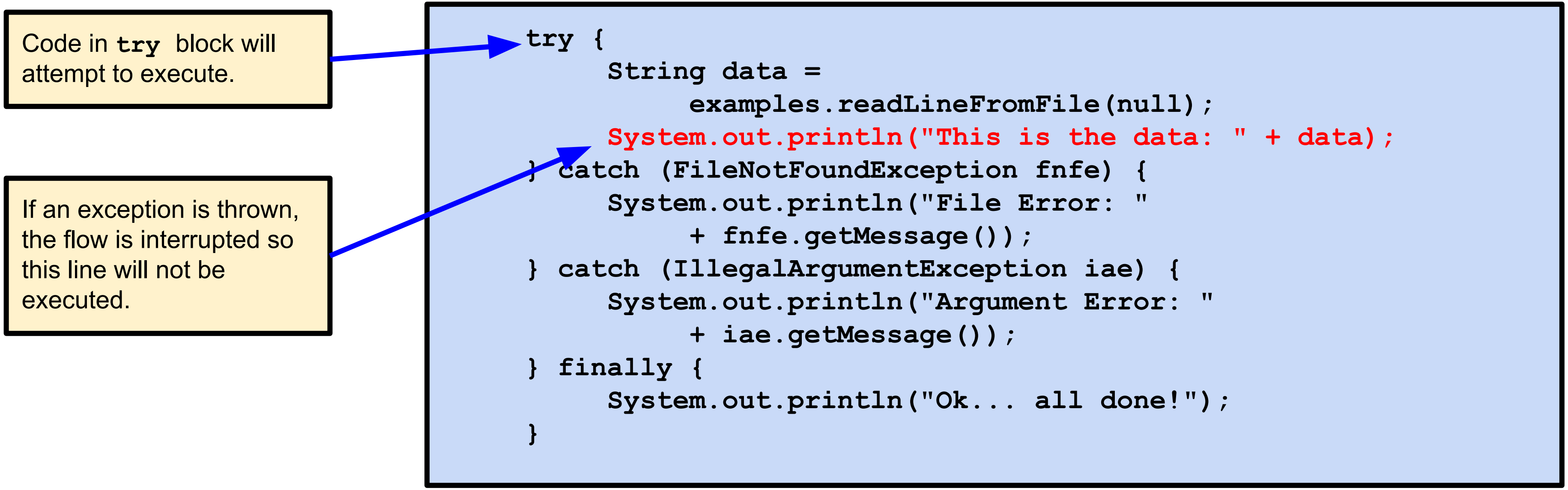
```java
try {
     String data =
          examples.readLineFromFile(null);
     System.out.println("This is the data: " + data);
} catch (FileNotFoundException fnfe) {
     System.out.println("File Error: "
          + fnfe.getMessage());
} catch (IllegalArgumentException iae) {
     System.out.println("Argument Error: "
          + iae.getMessage());
} finally {
     System.out.println("Ok... all done!");
}
```

A Closer Look at the Try-Catch-Finally Block

Code in **`try`** block will attempt to execute.

```java
try {
     String data =
          examples.readLineFromFile(null);
     System.out.println("This is the data: " + data);
} catch (FileNotFoundException fnfe) {
     System.out.println("File Error: "
          + fnfe.getMessage());
} catch (IllegalArgumentException iae) {
     System.out.println("Argument Error: "
          + iae.getMessage());
} finally {
     System.out.println("Ok... all done!");
}
```

A Closer Look at the Try-Catch-Finally Block

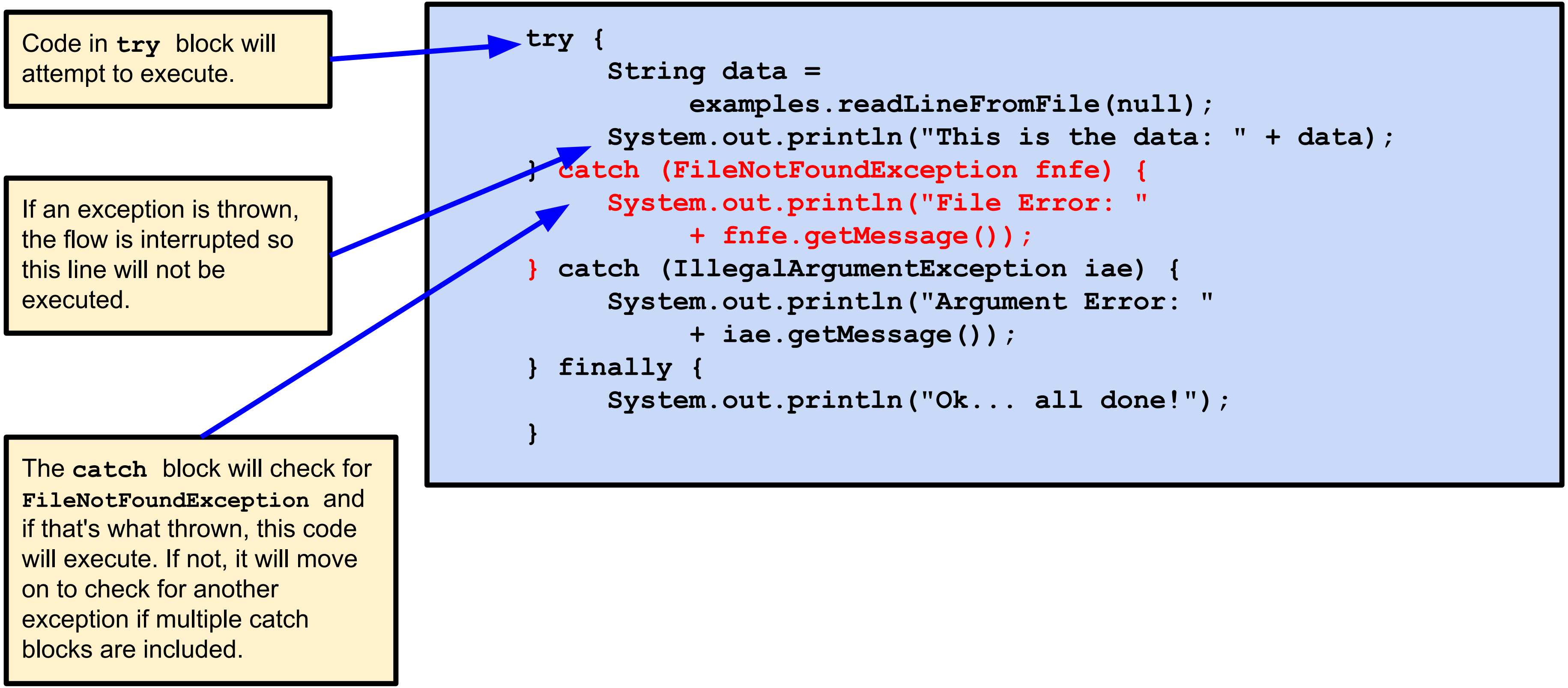


Code in **try** block will attempt to execute.

If an exception is thrown, the flow is interrupted so this line will not be executed.

```
try {
     String data =
          examples.readLineFromFile(null);
     System.out.println("This is the data: " + data);
} catch (FileNotFoundException fnfe) {
     System.out.println("File Error: "
          + fnfe.getMessage());
} catch (IllegalArgumentException iae) {
     System.out.println("Argument Error: "
          + iae.getMessage());
} finally {
     System.out.println("Ok... all done!");
}
```

A Closer Look at the Try-Catch-Finally Block



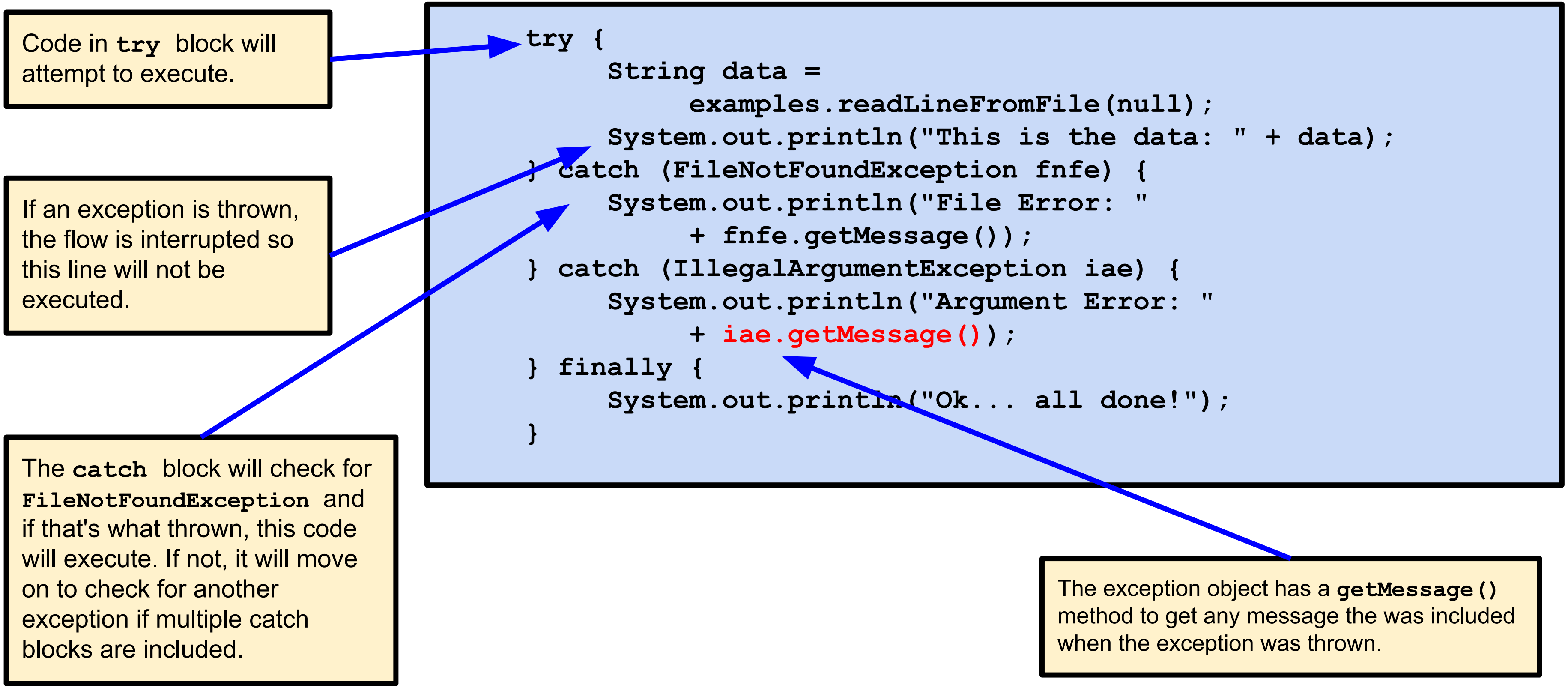
Code in **try** block will attempt to execute.

If an exception is thrown, the flow is interrupted so this line will not be executed.

The **catch** block will check for **FileNotFoundException** and if that's what thrown, this code will execute. If not, it will move on to check for another exception if multiple catch blocks are included.

```
try {
     String data =
          examples.readLineFromFile(null);
     System.out.println("This is the data: " + data);
} catch (FileNotFoundException fnfe) {
     System.out.println("File Error: "
          + fnfe.getMessage());
} catch (IllegalArgumentException iae) {
     System.out.println("Argument Error: "
          + iae.getMessage());
} finally {
     System.out.println("Ok... all done!");
}
```

A Closer Look at the Try-Catch-Finally Block



```
try {
        String data =
                examples.readLineFromFile(null);
        System.out.println("This is the data: " + data);
} catch (FileNotFoundException fnfe) {
        System.out.println("File Error: "
                + fnfe.getMessage());
} catch (IllegalArgumentException iae) {
        System.out.println("Argument Error: "
                + iae.getMessage());
} finally {
        System.out.println("Ok... all done!");
}
```

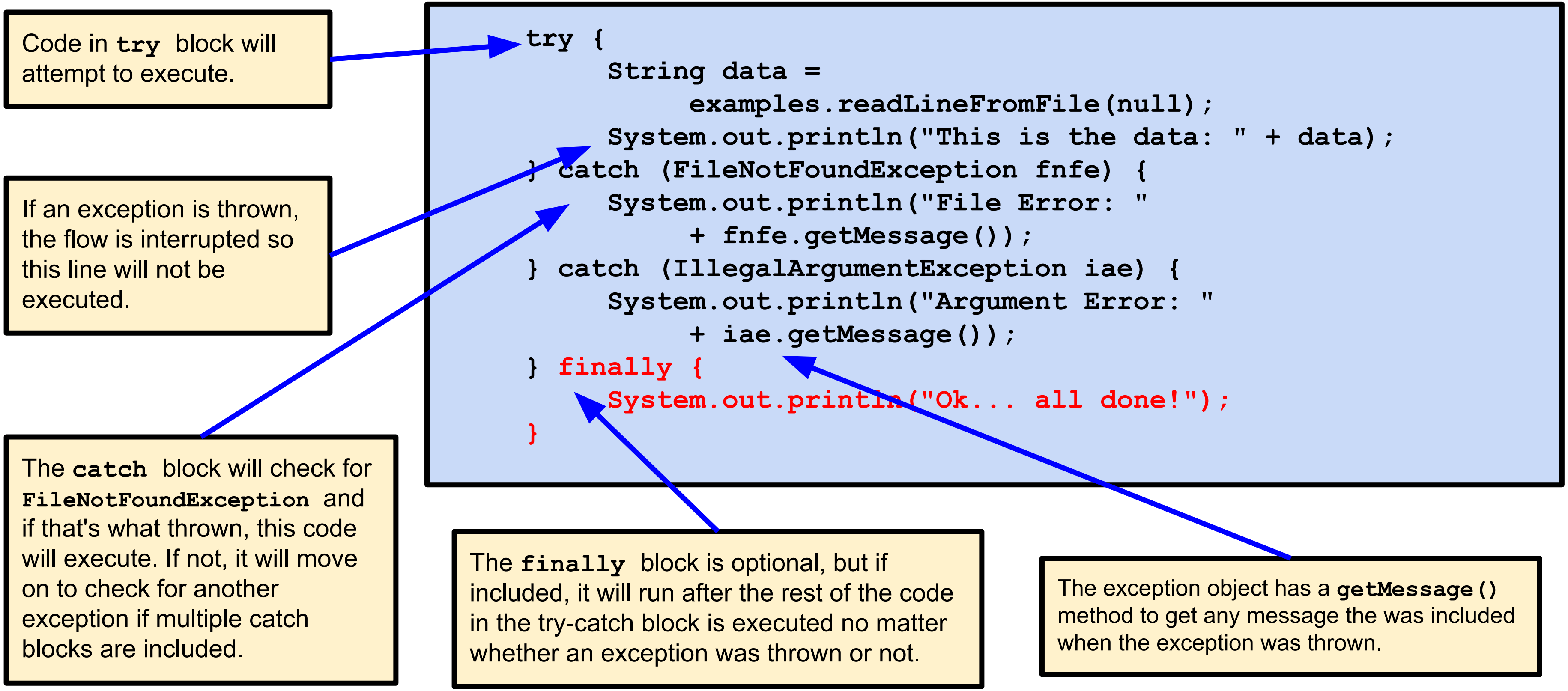
Code in **`try`** block will attempt to execute.

If an exception is thrown, the flow is interrupted so this line will not be executed.

The **`catch`** block will check for **`FileNotFoundException`** and if that's what thrown, this code will execute. If not, it will move on to check for another exception if multiple catch blocks are included.

The exception object has a **`getMessage()`** method to get any message the was included when the exception was thrown.

A Closer Look at the Try-Catch-Finally Block



```
try {
     String data =
          examples.readLineFromFile(null);
     System.out.println("This is the data: " + data);
} catch (FileNotFoundException fnfe) {
     System.out.println("File Error: "
          + fnfe.getMessage());
} catch (IllegalArgumentException iae) {
     System.out.println("Argument Error: "
          + iae.getMessage());
} finally {
     System.out.println("Ok... all done!");
}
```

Code in **`try`** block will attempt to execute.

If an exception is thrown, the flow is interrupted so this line will not be executed.

The **`catch`** block will check for **`FileNotFoundException`** and if that's what thrown, this code will execute. If not, it will move on to check for another exception if multiple catch blocks are included.

The **`finally`** block is optional, but if included, it will run after the rest of the code in the try-catch block is executed no matter whether an exception was thrown or not.

The exception object has a **`getMessage()`** method to get any message the was included when the exception was thrown.

A Few More Notes on Exceptions...

- SInce all exceptions are inherited from the **`Exception`** class, We can make use of polymorphism in our code.
    - We can catch many different kinds of exceptions with a single catch block (if it makes sense). For instance:
        - The FileNotFoundException we will learn about next inherits from IOException. We can handle all file related exceptions by handling IOException.
        - We can also write a general catch block which handles Exception to catch all exceptions. Doing this is discouraged though, as it defeats some of the purpose of having many different types of exceptions.

A Few More Notes on Exceptions...

- We can create our own exceptions to handle possible scenarios that arise in our code which we may want to throw exceptions for.

- When a runtime exception that we don't catch is thrown, we will see something called a stack trace. The stack trace shows you the whole stack of method calls that were made to get to the point where the exception was thrown and are invaluable for figuring where code went wrong.

    - The **`Exception`** object that is available in **`catch`** blocks, can be used to read, and even print, the stack trace.

File Input

File Input

Java has the ability to read in data stored in a text file. It is one of many forms of inputs available to Java:

- Command Line user input (we have covered this one)
- Through a relational database
- Through a web interface using the Spring framework
- Through an external API

File Input : The File Class

The file class is the Java class that encapsulates what it means to be a file containing data. This is an instantiation of a **`File`** object.

```
File inputFile = new File("testFile.txt");
```

File Input : The File Class

The file class is the Java class that encapsulates what it means to be a file containing data. This is an instantiation of a **`File`** object.

```
File inputFile = new File("testFile.txt");
```

The simplest form of the the File constructor takes a **`String`** indicating the path of the file to open.

File Input : The File Class

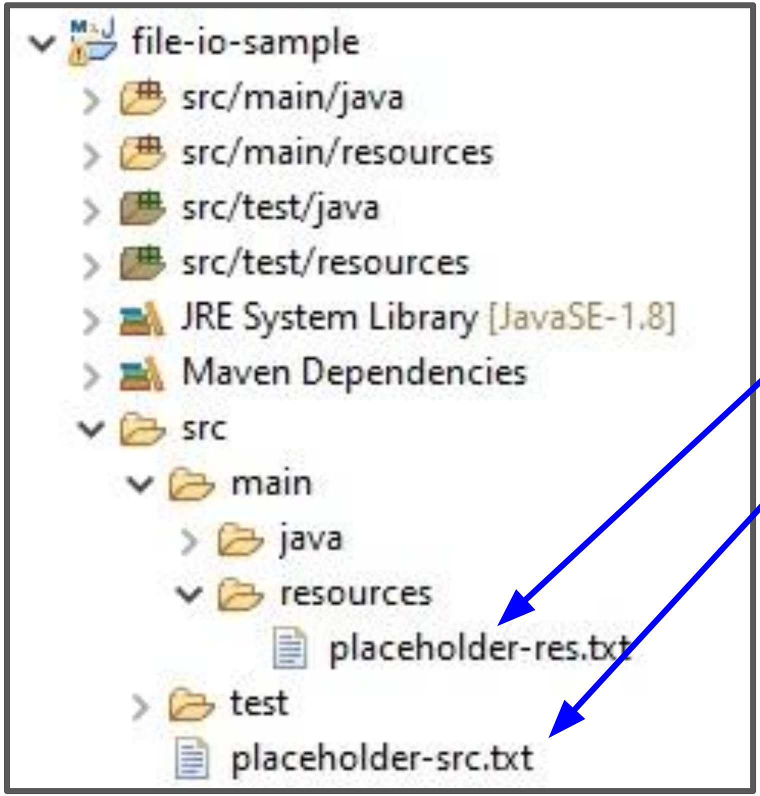
The file class is the Java class that encapsulates what it means to be a file containing data. This is an instantiation of a **File** object.

```
File inputFile = new File("testFile.txt");
```

The simplest form of the the File constructor takes a **String** indicating the path of the file to open.

The **File** class can be used to perform many actions on files or directories in the filesystem.

File Input : The File Class



- file-io-sample
  - src/main/java
  - src/main/resources
  - src/test/java
  - src/test/resources
  - JRE System Library [JavaSE-1.8]
  - Maven Dependencies
  - src
    - main
      - java
      - resources
        - placeholder-res.txt
    - test
    - placeholder-src.txt

The location in the File constructor is relative to the root of the Java project.

```
File resourcesfile =
      new File("src/main/resources/placeholder-res.txt");

File srcFile =
      new File("src/placeholder-src.txt");
```

File Input : The File Class Methods

There are two methods of the file class that are essential for file input:

- **`.exists()`**: Returns a boolean indicating whether a file exists. We would not want to proceed to parse a file if the file itself was missing!

- **`.getAbsoluteFile()`**: Returns the same File object you instantiated but with an absolute path. You can think of this as a getter. It returns a File object.

Data Streams

Methods exist to read all text in very quickly with one line of code. They pull the entire file into memory though.Typically, we don't want to do that, especially for large files. This is equivalent of sitting to watch a NetFlix movie and waiting for the entire movie to load before you start watching it!!!!

- A **stream** refers to a sequence of bytes that can read and write to some sort of backing data store.
- A stream is like an assembly line, where you process each thing as it comes through, in order.
- The **`Scanner`** class we used to read keyboard input is a way to read data streams.

File and Scanner

- A **`File`** object and a **`Scanner`** object can work in conjunction with one another to read file data.

- Once a **`File`** object exists, we can instantiate a **`Scanner`** object with the **`File`** as a constructor argument. Previously, we used System.in as the argument to indicate we were reading from the keyboard.

- Before we look at that though… let's review how we have been creating **`Scanner`** objects...

Review of Scanner

Let's review how we created a **`Scanner`** object previously:

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

System.out.print("Type something: ");
String data = input.nextLine();

System.out.println(data);
```

We created a new **`Scanner`** object and passed the keyboard input stream in the constructor.

Review of Scanner

Let's review how we created a **`Scanner`** object previously:

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

System.out.print("Type something: ");
String data = input.nextLine();

System.out.println(data);
```

We created a new **`Scanner`** object and passed the keyboard input stream in the constructor.

There is an issue with this code. When you open a **`Scanner`**, or any other resource that can be opened and closed, you should close it when you are done.

Not doing so can cause all kinds of issues (one example is that if you are using a **`File`** that is buffering data, the data may never get "flushed" and written to the filesystem if you don't close the **`File`**).

Review of Scanner

Let's review how we created a **`Scanner`** object previously:

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

System.out.print("Type something: ");
String data = input.nextLine();

System.out.println(data);
```

We created a new **`Scanner`** object and passed the keyboard input stream in the constructor.

There is an issue with this code. When you open a **`Scanner`**, or any other resource that can be opened and closed, you should close it when you are done.

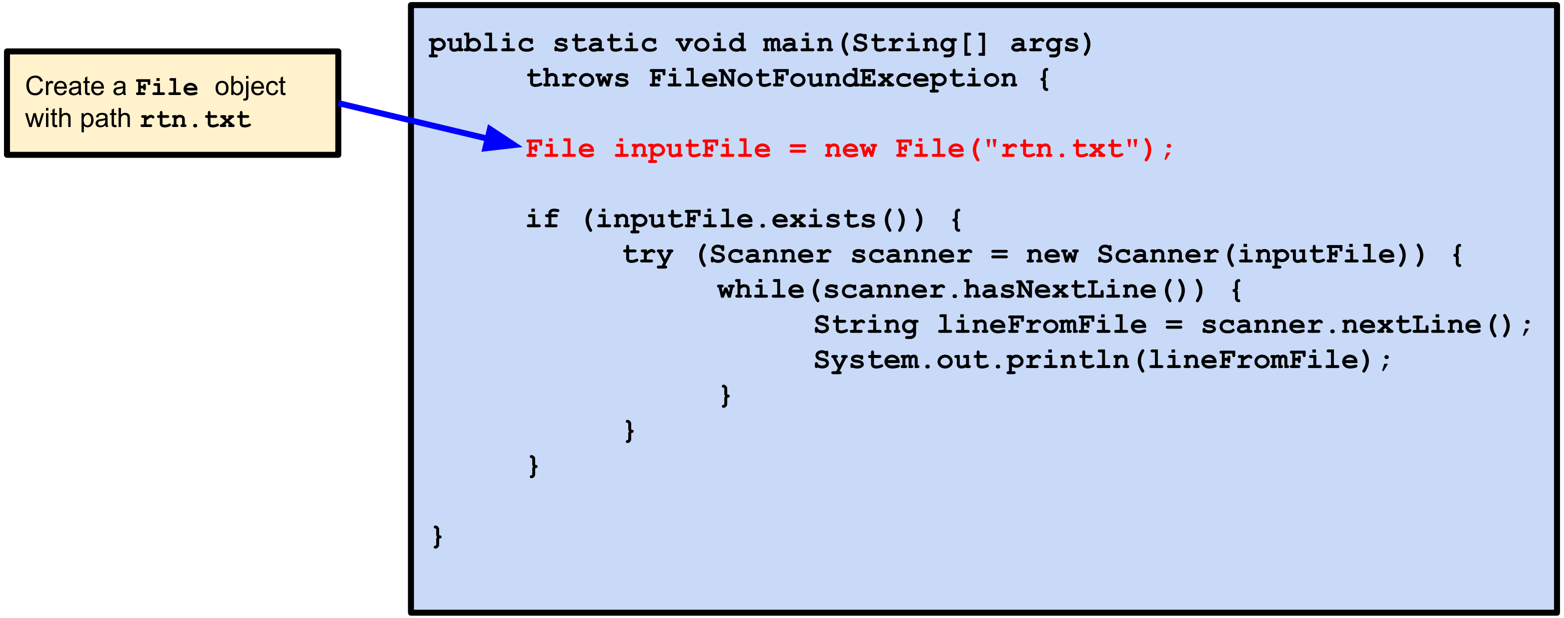
Not doing so can cause all kinds of issues (one example is that if you are using a **`File`** that is buffering data, the data may never get "flushed" and written to the filesystem if you don't close the **`File`**).

But wait…. what happens if we call **`input.close()`** ? If you do this you will find your program **can no longer read from the keyboard!!!** Why? Because you closed the input stream associated with keyboard when you closed the **`Scanner`**!!! **`System.in`** is a special stream that **shouldn't be closed** (it will be closed when your program exits), but other inputs used with **`Scanner`** should always be closed.

File + Scanner

```
public static void main(String[] args)
      throws FileNotFoundException {

      File inputFile = new File("rtn.txt");

      if (inputFile.exists()) {
            try (Scanner scanner = new Scanner(inputFile)) {
                  while(scanner.hasNextLine()) {
                        String lineFromFile = scanner.nextLine();
                        System.out.println(lineFromFile);
                  }
            }
      }

}
```

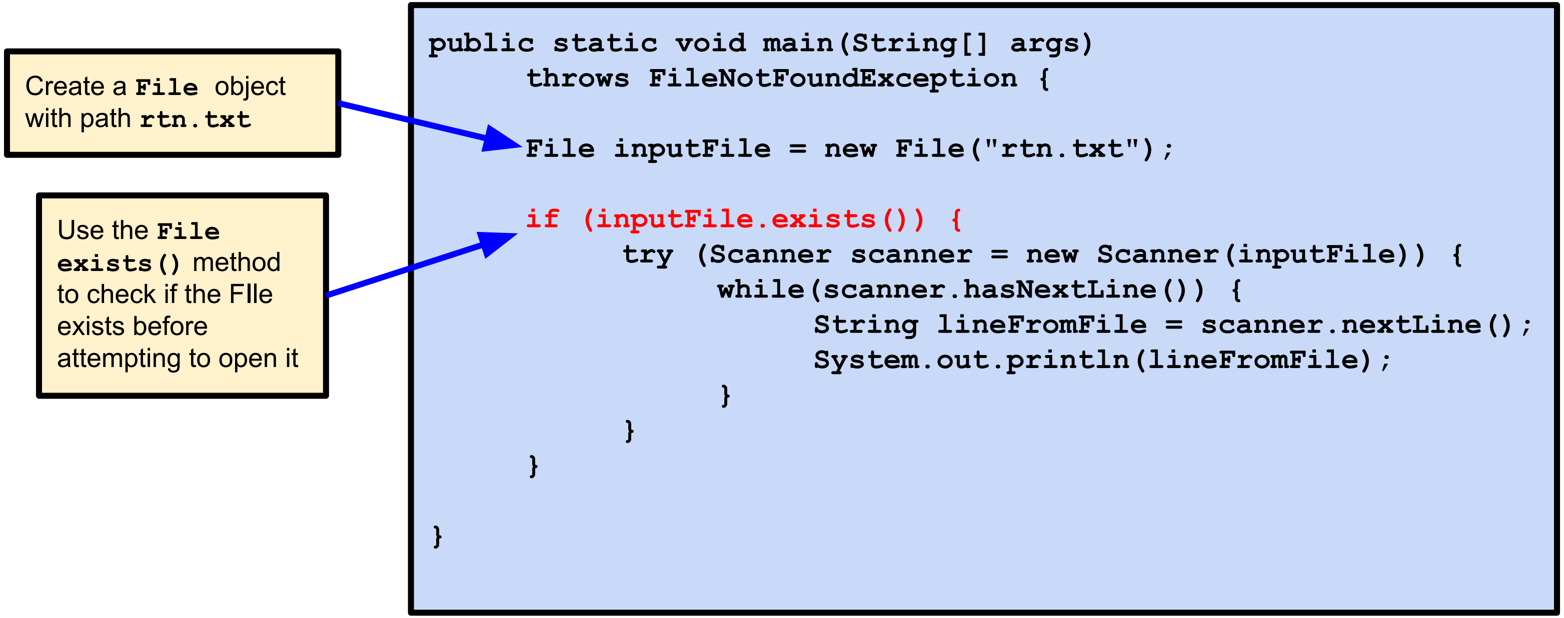
File + Scanner



Create a **File** object with path **rtn.txt**

```java
public static void main(String[] args)
      throws FileNotFoundException {

      File inputFile = new File("rtn.txt");

      if (inputFile.exists()) {
            try (Scanner scanner = new Scanner(inputFile)) {
                  while(scanner.hasNextLine()) {
                        String lineFromFile = scanner.nextLine();
                        System.out.println(lineFromFile);
                  }
            }
      }

}
```

File + Scanner

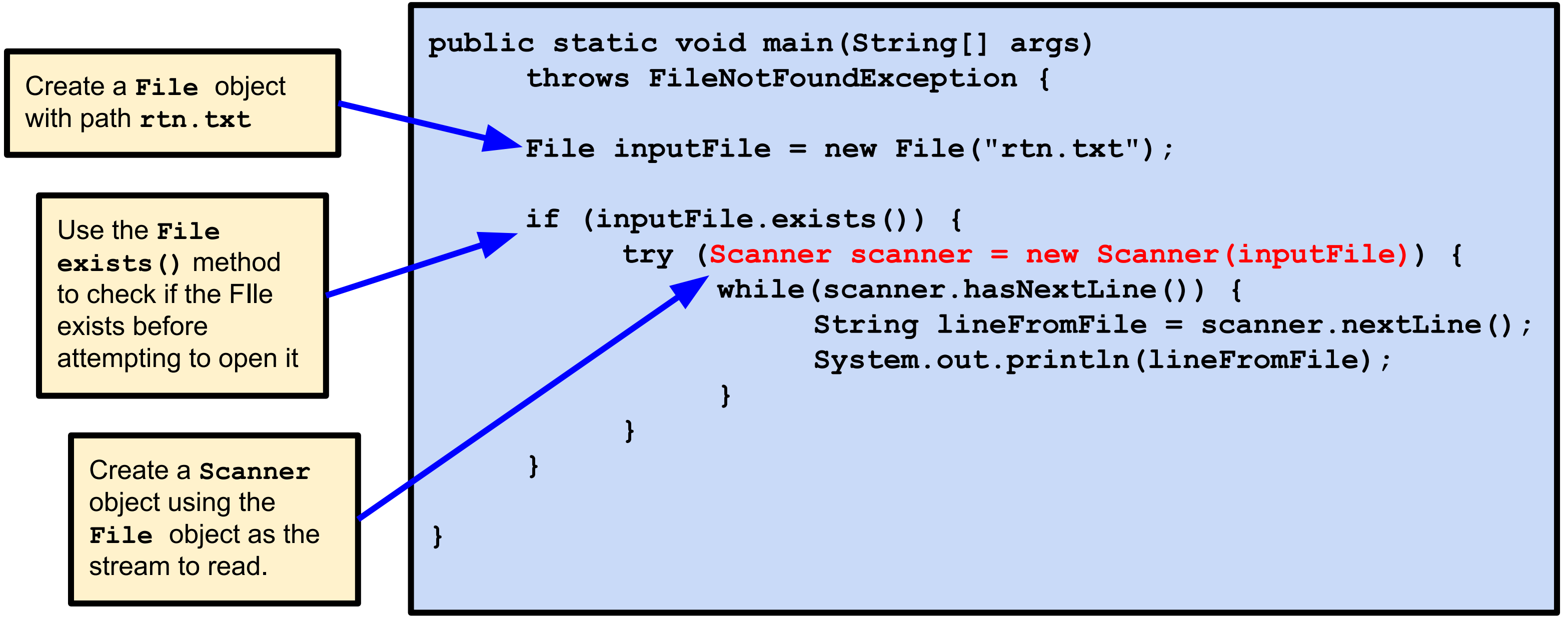


Create a **File** object with path **rtn.txt**

Use the **File** **exists()** method to check if the FIle exists before attempting to open it

```
public static void main(String[] args)
     throws FileNotFoundException {

     File inputFile = new File("rtn.txt");

     if (inputFile.exists()) {
          try (Scanner scanner = new Scanner(inputFile)) {
               while(scanner.hasNextLine()) {
                    String lineFromFile = scanner.nextLine();
                    System.out.println(lineFromFile);
               }
          }
     }

}
```

File + Scanner



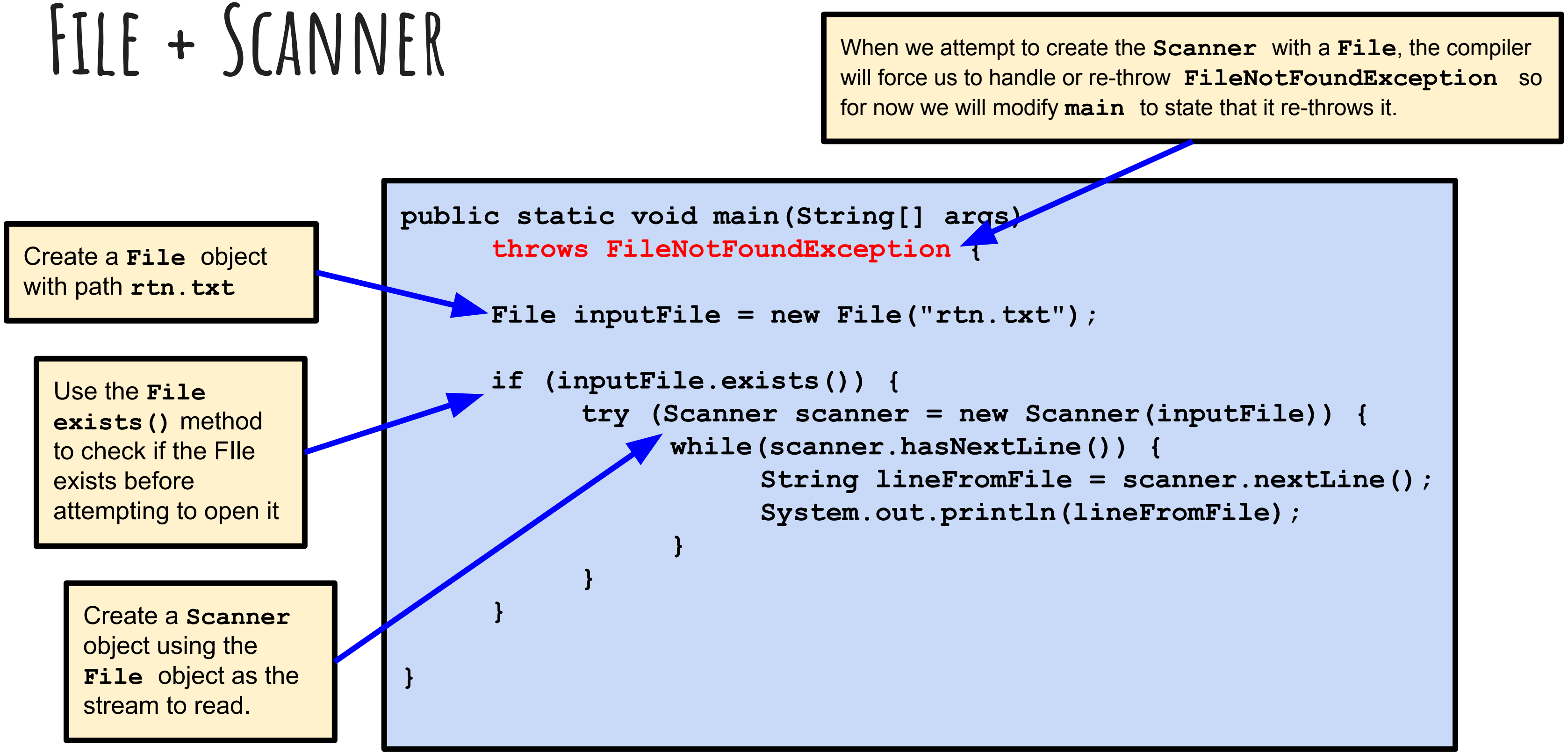
Create a **File** object with path **rtn.txt**

Use the **File** **exists()** method to check if the FIle exists before attempting to open it

Create a **Scanner** object using the **File** object as the stream to read.

```
public static void main(String[] args)
      throws FileNotFoundException {

      File inputFile = new File("rtn.txt");

      if (inputFile.exists()) {
            try (Scanner scanner = new Scanner(inputFile)) {
                  while(scanner.hasNextLine()) {
                        String lineFromFile = scanner.nextLine();
                        System.out.println(lineFromFile);
                  }
            }
      }

}
```

File + Scanner



When we attempt to create the **`Scanner`** with a **`File`**, the compiler will force us to handle or re-throw **`FileNotFoundException`** so for now we will modify **`main`** to state that it re-throws it.

Create a **`File`** object with path **`rtn.txt`**

Use the **`File`** **`exists()`** method to check if the FIle exists before attempting to open it

Create a **`Scanner`** object using the **`File`** object as the stream to read.

```
public static void main(String[] args)
    throws FileNotFoundException {

    File inputFile = new File("rtn.txt");

    if (inputFile.exists()) {
        try (Scanner scanner = new Scanner(inputFile)) {
            while(scanner.hasNextLine()) {
                String lineFromFile = scanner.nextLine();
                System.out.println(lineFromFile);
            }
        }
    }

}
```

File + Scanner

When we attempt to create the **`Scanner`** with a **`File`**, the compiler will force us to handle or re-throw **`FileNotFoundException`** so for now we will modify **`main`** to state that it re-throws it.

Create a **`File`** object with path **`rtn.txt`**

Use the **`File`** **`exists()`** method to check if the FIle exists before attempting to open it

Create a **`Scanner`** object using the **`File`** object as the stream to read.

```
public static void main(String[] args)
          throws FileNotFoundException {

          File inputFile = new File("rtn.txt");

          if (inputFile.exists()) {
                    try (Scanner scanner = new Scanner(inputFile)) {
                              while(scanner.hasNextLine()) {
                                        String lineFromFile = scanner.nextLine();
                                        System.out.println(lineFromFile);
                              }
                    }
          }

}
```

Did you notice that the **`Scanner`** was created as a parameter of a **`try`** block? What's that about? Stay tuned...

File + Scanner

When we attempt to create the **Scanner** with a **File**, the compiler will force us to handle or re-throw **FileNotFoundException** so for now we will modify **main** to state that it re-throws it...

```
public static void main(String[] args)
          throws FileNotFoundException {

          File inputFile = new File("rtn.txt");

          if (inputFile.exists()) {
                    try (Scanner scanner = new Scanner(inputFile)) {
                              while(scanner.hasNextLine()) {
                                        String lineFromFile = scanner.nextLine();
                                        System.out.println(lineFromFile);
                              }
                    }
          }

}
```

Create a **File** object with path **rtn.txt**

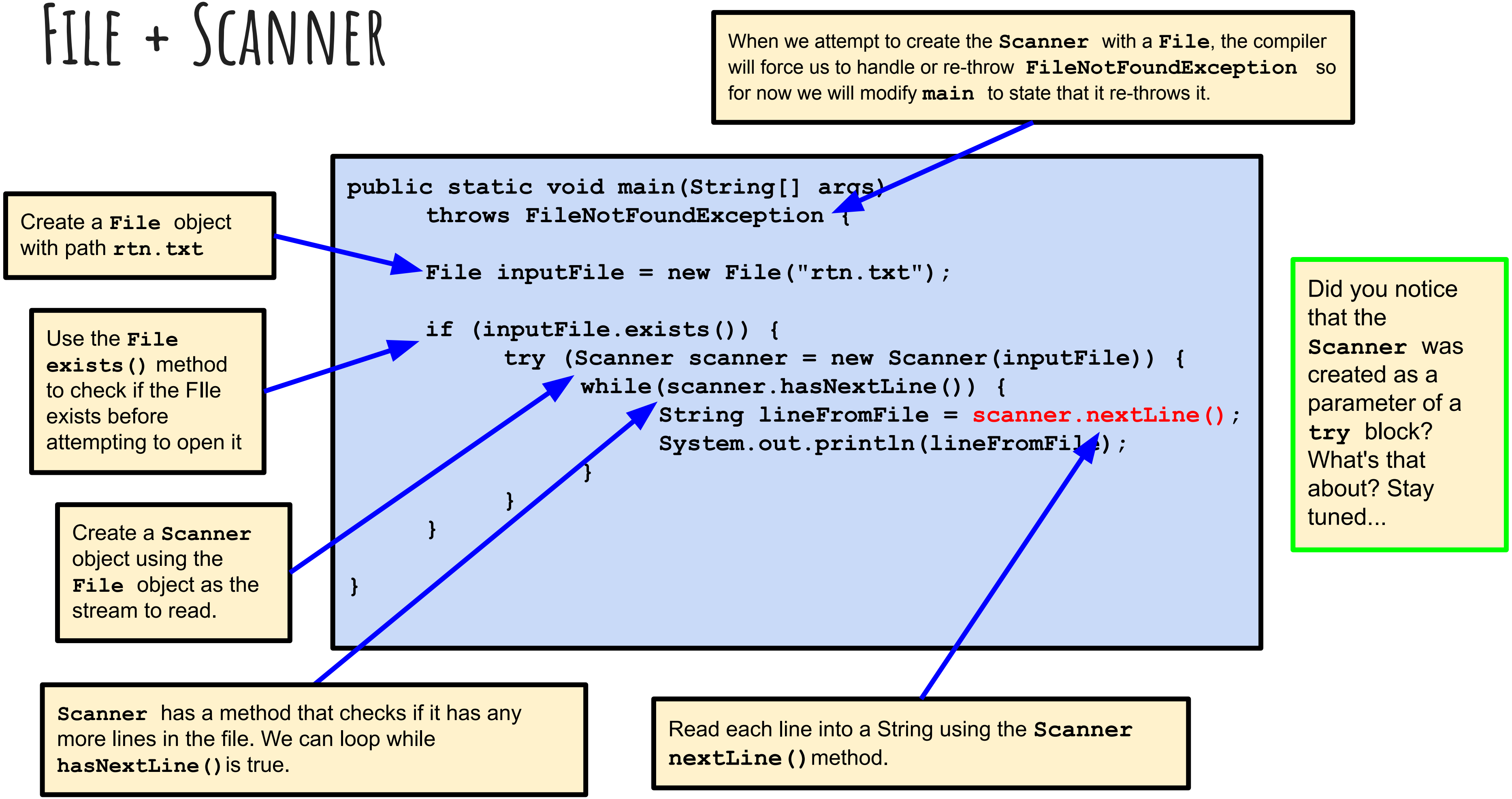
Use the **File** **exists()** method to check if the FIle exists before attempting to open it

Create a **Scanner** object using the **File** object as the stream to read.

Scanner has a method that checks if it has any more lines in the file. We can loop while **hasNextLine()** is true.

Did you notice that the **Scanner** was created as a parameter of a **try** block? What's that about? Stay tuned...

File + Scanner



When we attempt to create the **`Scanner`** with a **`File`**, the compiler will force us to handle or re-throw **`FileNotFoundException`** so for now we will modify **`main`** to state that it re-throws it...

```
public static void main(String[] args)
      throws FileNotFoundException {

      File inputFile = new File("rtn.txt");

      if (inputFile.exists()) {
            try (Scanner scanner = new Scanner(inputFile)) {
                  while(scanner.hasNextLine()) {
                        String lineFromFile = scanner.nextLine();
                        System.out.println(lineFromFile);
                  }
            }
      }

}
```

Create a **`File`** object with path **`rtn.txt`**

Use the **`File`** **`exists()`** method to check if the FIle exists before attempting to open it

Create a **`Scanner`** object using the **`File`** object as the stream to read.

Did you notice that the **`Scanner`** was created as a parameter of a **`try`** block? What's that about? Stay tuned...

`Scanner` has a method that checks if it has any more lines in the file. We can loop while **`hasNextLine()`** is true.

Read each line into a String using the **`Scanner`** **`nextLine()`** method.

The Try With Resources Block

Now let's address that weird way Scanner was created as a parameter of a **`try`** block...

```
try (Scanner scanner = new Scanner(inputFile)) {
     while(scanner.hasNextLine()) {
         String lineFromFile = scanner.nextLine();
         System.out.println(lineFromFile);
     }
}
```

The Try With Resources Block

Now let's address that weird way Scanner was created as a parameter of a **`try`** block...

```
try (Scanner scanner = new Scanner(inputFile)) {
     while(scanner.hasNextLine()) {
          String lineFromFile = scanner.nextLine();
          System.out.println(lineFromFile);
     }
}
```

In the past, opening and closing resources often involved repeating code to make sure that resources got closed whether the code ran as expected or wound up in a **`catch`** block due to an exception.

The **try-with-resources** version of a **`try`** block was created to handle this. If you create a resource as a parameter of a **`try`** block, the resource will be closed as soon as the **`try`** block is exited, whether that is through normal flow or when an exception is thrown and the code jumps to a **`catch`** block.

The Try With Resources Block

Now let's address that weird way Scanner was created as a parameter of a **`try`** block...

```
try (Scanner scanner = new Scanner(inputFile)) {
     while(scanner.hasNextLine()) {
          String lineFromFile = scanner.nextLine();
          System.out.println(lineFromFile);
     }
}
```

In the past, opening and closing resources often involved repeating code to make sure that resources got closed whether the code ran as expected or wound up in a **`catch`** block due to an exception.

The **<u>try-with-resources</u>** version of a **`try`** block was created to handle this. If you create a resource as a parameter of a **`try`** block, the resource will be closed as soon as the **`try`** block is exited, whether that is through normal flow or when an exception is thrown and the code jumps to a **`catch`** block.

Note that a try-with-resources block can be used to provide this "auto-closing" mechanism even in scenarios where it is not necessary to catch an exception. In this special case, the try block does not always require a **`catch`** or **`finally`** clause.

Catching FileNotFoundExceptions

This is an example of handling the possibility of a **`FileNotFoundException`** when opening a **`File`** rather than having the method pass the buck up the chain.

Here we use the try-with-resources block to create the **`Scanner`** resource and add a **`catch`** block to handle the possible exception.

Note that the **`main`** method no longer re-throws the exception.

```
public static void main(String[] args)
                  {

    File inputFile = new File("rtn.txt");

    if (inputFile.exists()) {
        try (Scanner scanner = new Scanner(inputFile)) {
            while(scanner.hasNextLine()) {
                String lineFromFile = scanner.nextLine();
                System.out.println(lineFromFile);
            }
        } catch (FileNotFoundException e) {
            System.out.println("File not found.");
        }
    }
}
```

Exceptions When Reading Streams

Exceptions can often occur when reading streams. Here are some example of common scenarios when I/O Exceptions might occur:

- Directory not found
- End of stream reached
- File not found
- Path too long (Windows only)

What is File I/O Used for In the Real World?

Here are just a few examples of when you might read or write files in you future career:

- Importing Bulk Data Sets
- Desktop Applications - Reading in Configuration Settings
- Video Games - Data File
- Transmitting data to other systems