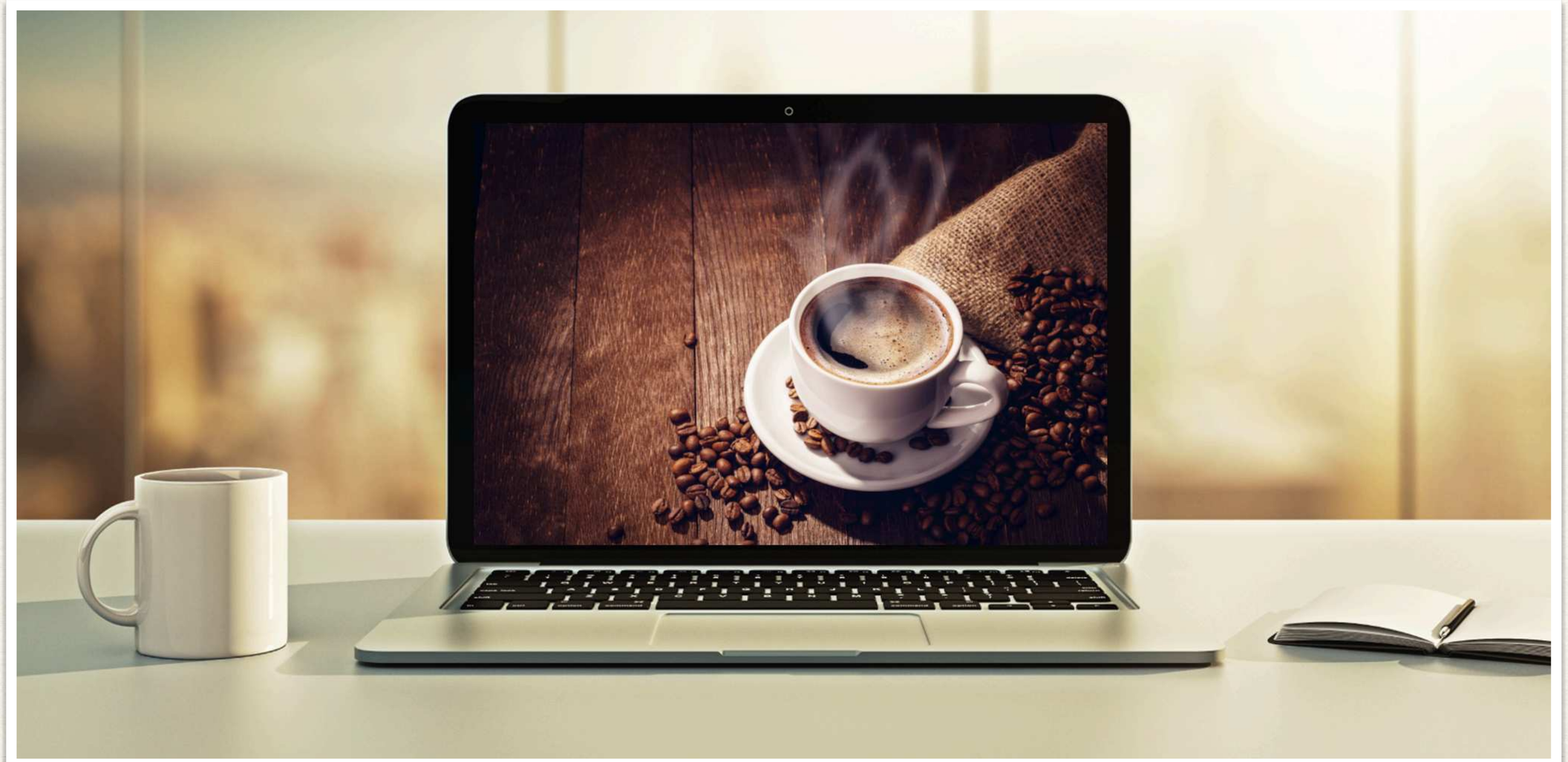
Exception Handling



What is exception handling?

What is Exception Handling?

- Exception Handling is used to handle the runtime errors so that the normal flow of an application can remain.
- Exception Handling is an event that occurs during execution of the code.
- This happens when the code / software runs into a condition that it was not expected and does not know how to handle this error

Lets go see top Java Runtime Errors examples

NullPointerExceptions

- Occurs when you try and access a property that is null.

```
String name = null;
name.length(); // Throws NullPointerException
```

ArrayIndexOutOfBoundsException

- Occurs when you try and access an array element using an index that does not exist.

```
String[] names = {“Eric”, “Chád”, “Milo”};
names[3]; // Throws ArrayIndexOutOfBoundsException
```

NumberFormatException

- Occurs when you try to convert a String to an int that does not represent a valid number

```
String notValidNumber = “Not a number”;
Integer.parseInt(notValidNumber); // Throws NumberFormatException
```

ArithmeticException

- Occurs when an invalid arithmetic is used

```
int number = 20 / 0; // Throws ArithmeticException
```

How do you create exception handling?

Try / Catch Block

- Allows the code to ‘try’ and execute, what could potentially throw an exception
- If an exception occurs the code will ‘catch’ the exception to prevent the application from crashing

```
try {
    // Try specific code
} catch(ExceptionType e) {
    // Catch and run different code if application crashes
}
```

Try / Catch Block Example

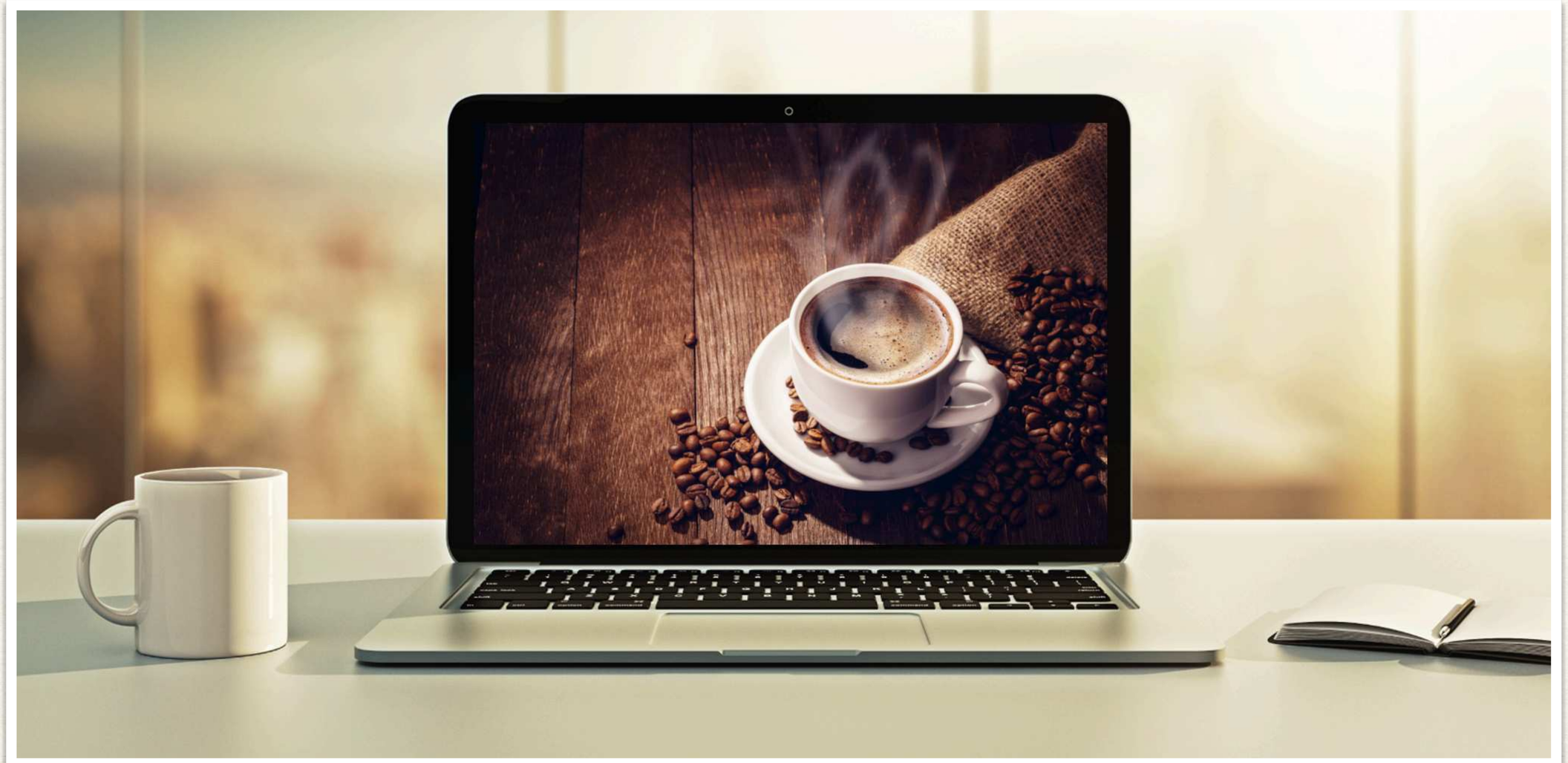
```
String numberString = “luv2code”;

int numberInt = Integer.parseInt(numberString);
```

Application will crash

```
String numberString = “luv2code”;

try {
    int numberInt = Integer.parseInt(numberString);
 } catch(ExceptionType e) {
     System.out.println(“You cannot parse this String”);
}
```

Application will NOT crash

Multiple Catch Statements



Multiple Catch Statements

```
parseString(null);
```

Something went wrong!

String needs to be valid int

```
public static void parseString(String numberString) {

  try {
      int numberInt = Integer.parseInt(numberString);
   }
   catch(NullPointerException e) {
       System.out.println("String needs to be valid int");
   }
   catch(NumberFormatException e) {
       System.out.println("String needs to be valid int");
   }
   catch(Exception e) {
       System.out.println("Something went wrong!");
  }
}
```

DUPLICATES!

Multiple Catch Statements

```
parseString(null);
```

String needs to be a valid int

```
public static void parseString(String numberString) {

  try {
      int numberInt = Integer.parseInt(numberString);
   }
   catch(NullPointerException e) {
       System.out.println("String needs to be valid int");
   }
   catch(NumberFormatException e) {
       System.out.println("String needs to be valid int");
   }
   catch(Exception e) {
       System.out.println("Something went wrong!");
  }
}
```

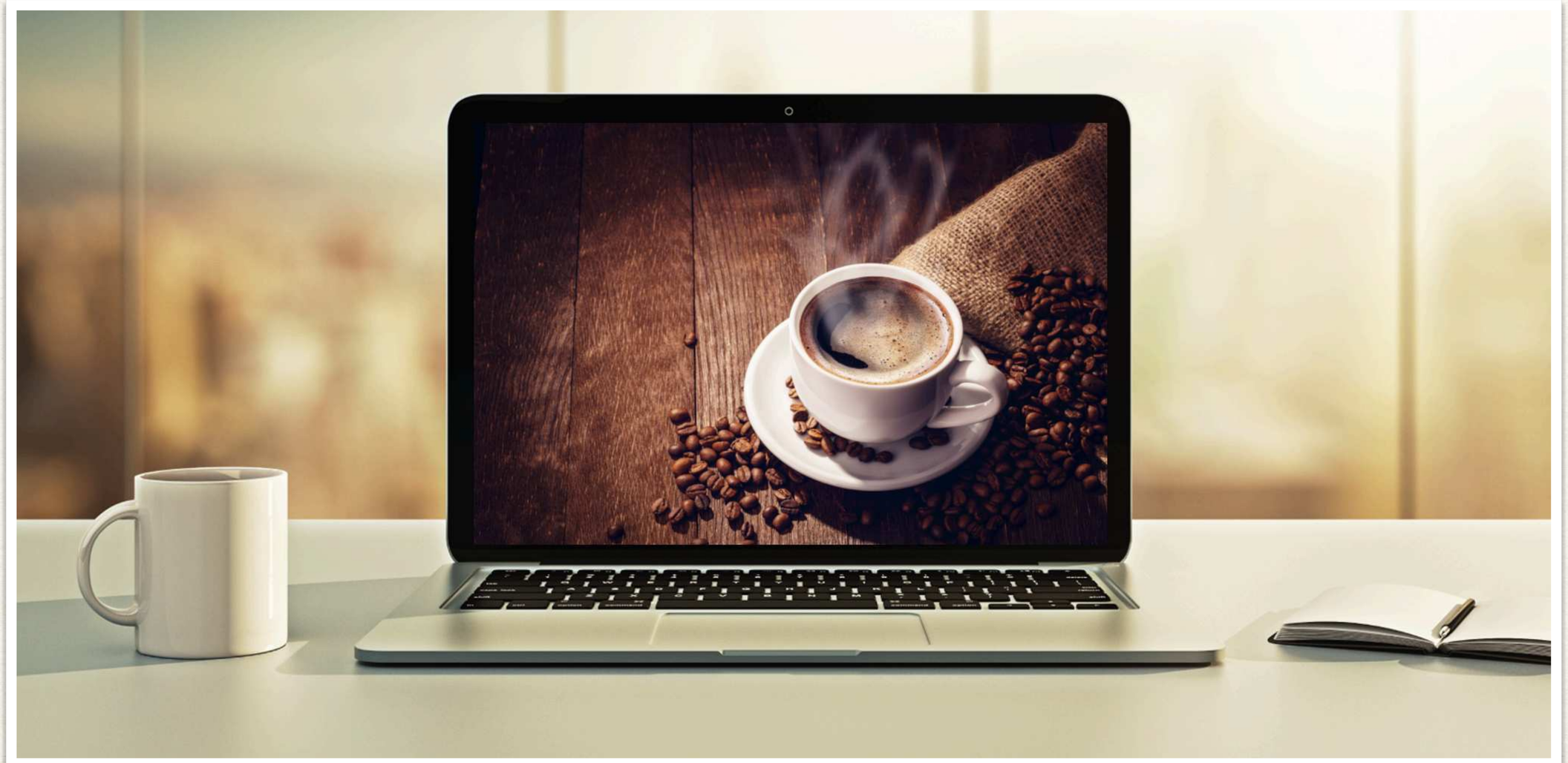
DUPLICATES!

Multiple Catch Statements

```
parseString(null);
```

```
public static void parseString(String numberString) {

  try {
      int numberInt = Integer.parseInt(numberString);
   }
   catch(NullPointerException | NumberFormatException e) {
       System.out.println("String needs to be valid int");
   }

  catch(Exception e) {
       System.out.println("Something went wrong!");
  }
}
```

Finally Block



What is the Finally Block?

- Will always execute after a Try or a Catch block
  - Always will run
- Used to clean up code
  - Closing file connections
  - Closing database connections

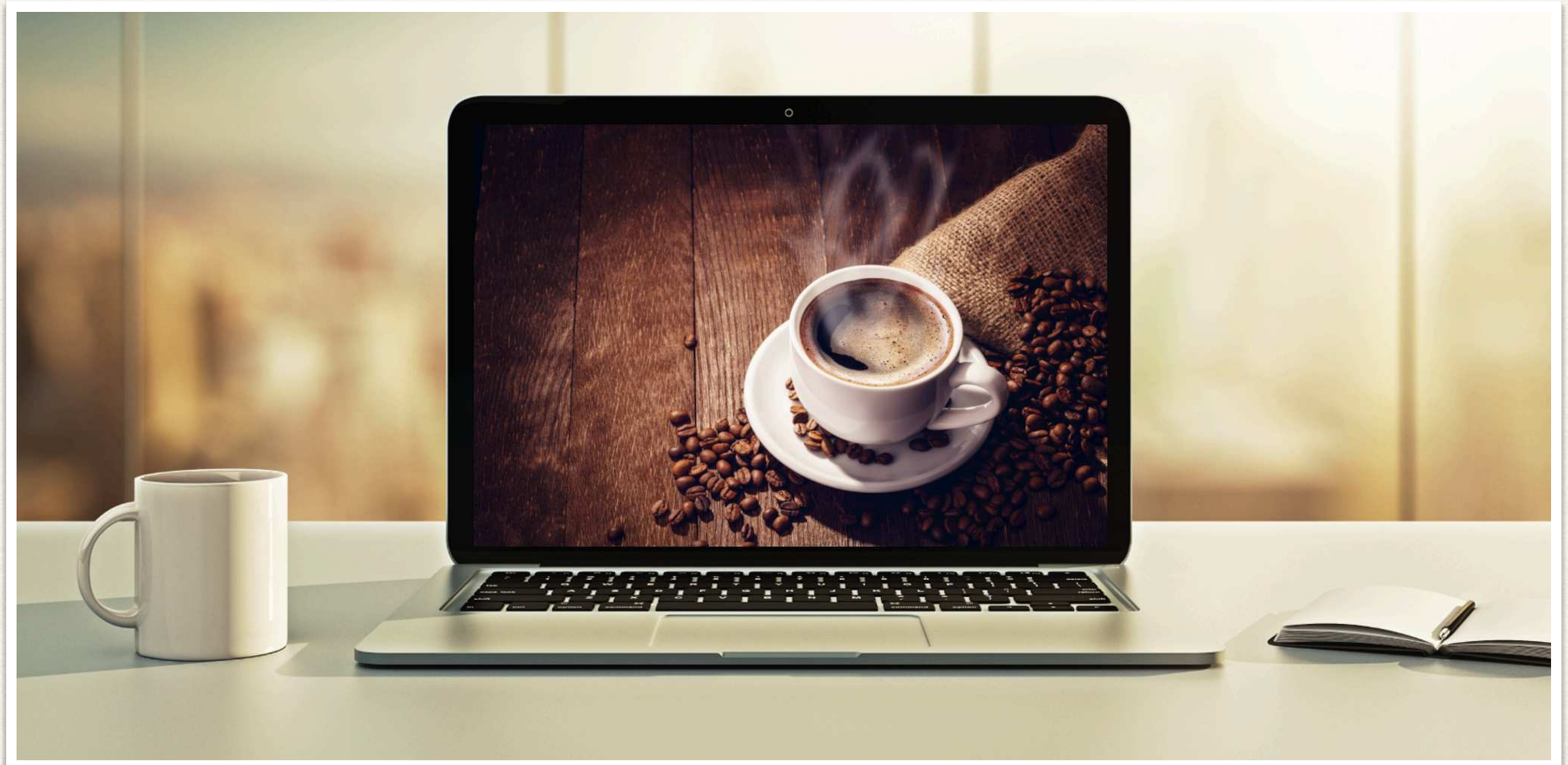
Multiple Catch Statements

```
parseString(null);
```

```
parseString("100");
```

```
public static void parseString(String numberString) {

  try {
      int numberInt = Integer.parseInt(numberString);
  }
  catch(NullPointerException | NumberFormatException e) {
       System.out.println("String needs to be valid int");
  }
  catch(Exception e) {
       System.out.println("Something went wrong!");
  }
  finally {
     System.out.println("Finally will always run");
  }
}
```

File Input (I/O) & Checked Exceptions



File Input & Checked Exceptions

- To start reading files in Java we will be using the FileReader
- A new FileReader requires a file as an argument
  - FileReader is a checked exception in Java
    - Java requires you to create exceptions at compile instead of runtime

```
FileReader fileReader = new FileReader(file.txt);
```

Red line due to unhandled exception

File Input & Checked Exceptions

Wrap the FileReader in a Try / Catch

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

  try {
      FileReader fileReader = new FileReader(file.txt);
  }
  catch(FileNotFoundException e) {
       System.out.println("File does not exist");
  }
}
```

Use the throws keyword

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

   FileReader fileReader = new FileReader(file.txt);
}
```

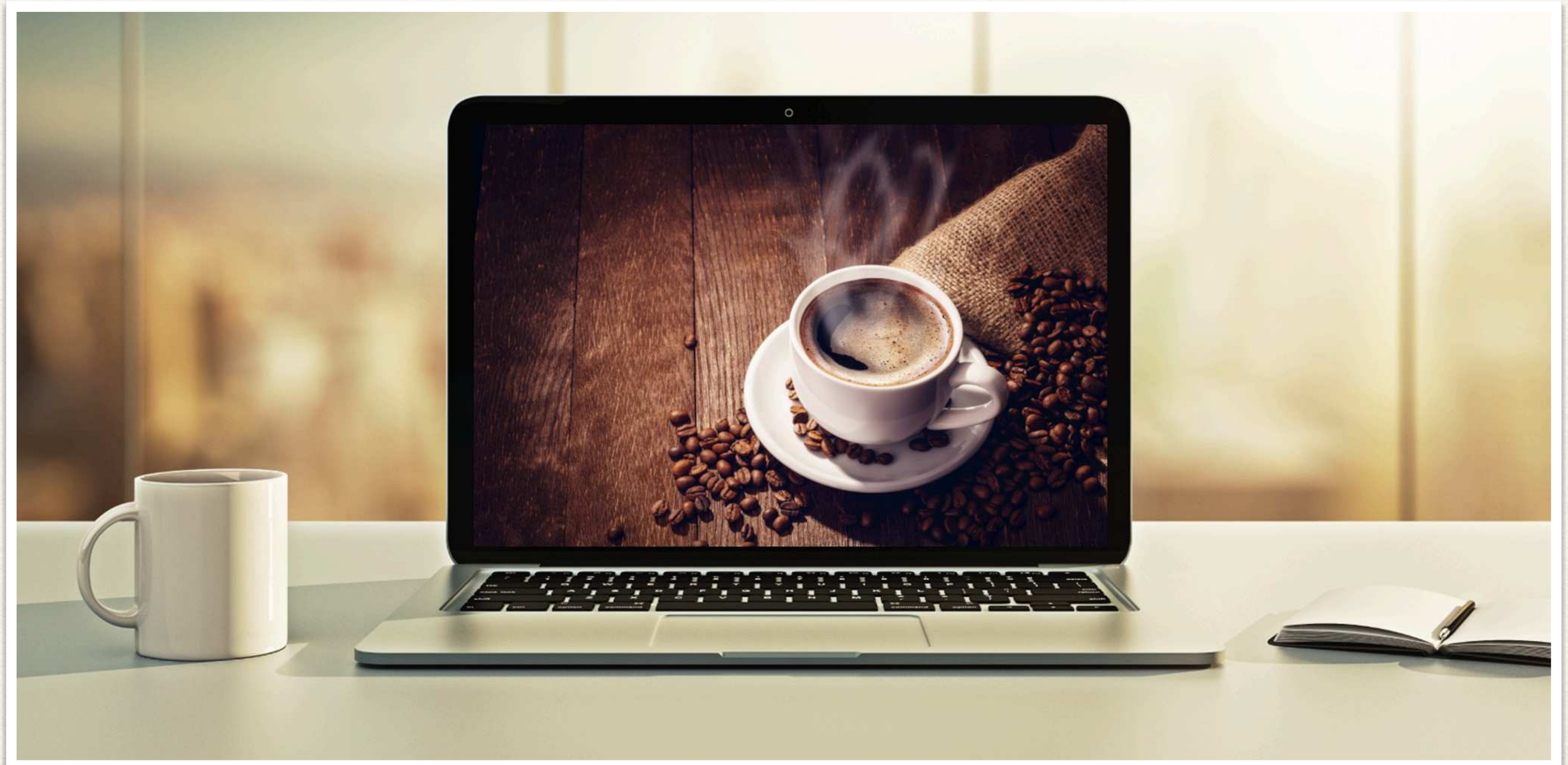
Throws?

- Used in the signature of a method
  - Listing the method as something that can throw an exception
- The caller of this method MUST handle the exception.
- Throws is for throwing the exception up to the parent (the part of the application that is calling this method that may have an exception)
- Checked Exceptions are exceptions where Java knows something can go wrong. *For example: finding a file for the FileReader*

File Input & Checked Exceptions

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

  try {
      readFile();
  }
  catch(FileNotFoundException e) {
       System.out.println("File does not exist");
  }
}

public static void readFile() throws FileNotFoundException {
   FileReader fileReader = new FileReader(file.txt);
}
```

FileReader



FileReader

- Used to read a file from a disk drive
- Directly reads the data from the character stream that comes from a file
  - Reads files character by character
  - Each time it reads a character it accesses the disk drive
  - Fairly resource intensive
  - Better alternatives (that we will get to later in this course)

```
FileReader fileReader = new FileReader(file.txt);
```

FileReader

Wrap the FileReader in a Try / Catch

Parent class of FileNotFoundException

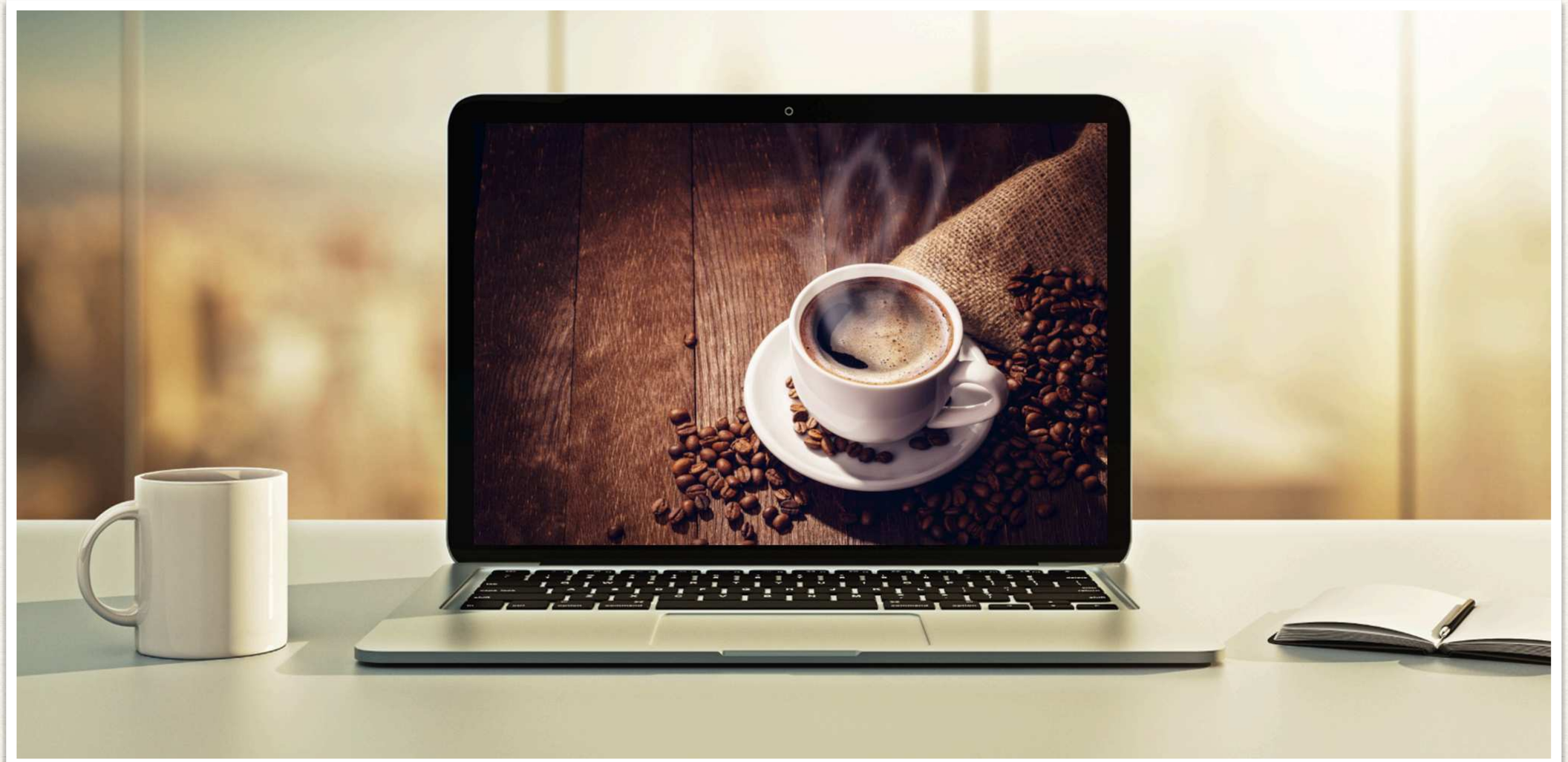
```
public static void main(String[] args) throws IOException {
   FileReader fileReader = null;

  try {
   // Connects to a new FileReader
      fileReader = new FileReader(story.txt);

   // -1 means end of file in a FileReader
      int character = 0;

   // Cast the unicode (UTF8) int into a char
      while ((character = fileReader.read()) != -1) {
        System.out.println((char) character);
   }
  }
 catch(FileNotFoundException e) {
      << Next Slide >>
  }
```

FileReader

Wrap the FileReader in a Try / Catch

```
<< Code Earlier >>

 catch(FileNotFoundException e) {
     System.out.println("File not found");
 }
 finally {
   if (fileReader != null) {

     // Close the file connection to get resources back
     fileReader.close();
   }
  }
}
```

BufferedReader



BufferedReader

- Used to read a file from a disk drive
- Buffers characters to provide efficient reading of characters, arrays, and line
  - Greatly improves improvement and resources
  - Minimize I/O compared to a FileReader alone
  - Gets multiple characters at a time and puts them in an internal buffer

```
BufferedReader reader = new BufferedReader(new FileReader(file.txt));
```

BufferedReader

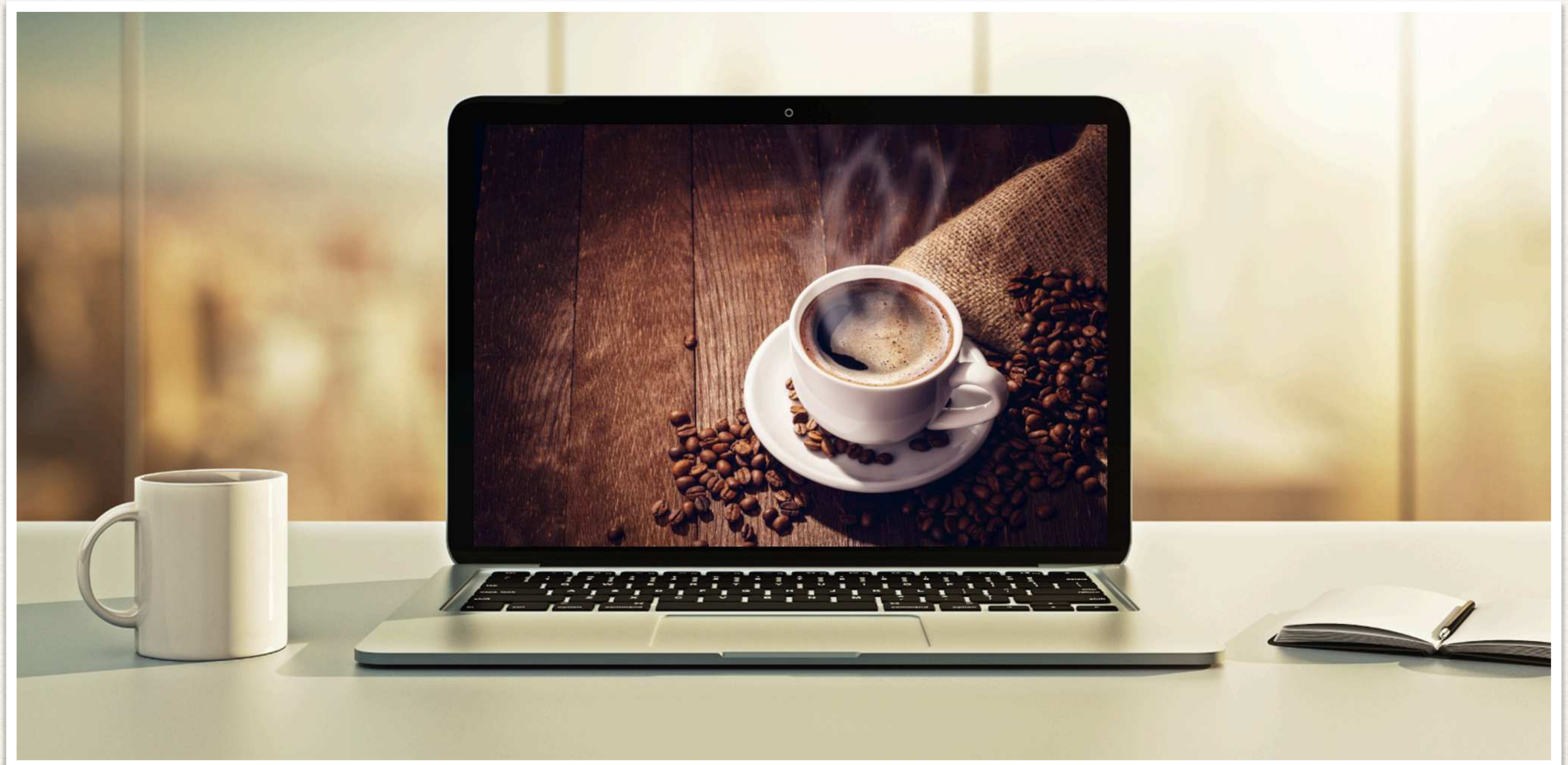
Wrap the BufferedReader in a Try / Catch

```
public static void main(String[] args) throws IOException {
   BufferedReader reader = null;

  try {
   // Connects to a new BufferedReader
      reader = new BufferedReader(new FileReader(story.txt));

      String line;

   // Fetches a line at a time
      while ((line = reader.readLine()) != null) {
        System.out.println(line);
   }
      reader.close();
  }
 catch(FileNotFoundException e) {
      e.printStackTrace();
  }
```

PrintStackTrace

- Method of throwable
- Prints the throwable (exception) along with:
  - Line number
  - Class name
- Useful for diagnosing exceptions within your application
- When dealing with large applications where you may have multiple exceptions this will help you identify the place and location of the exception

Try with Resources



Try with Resources

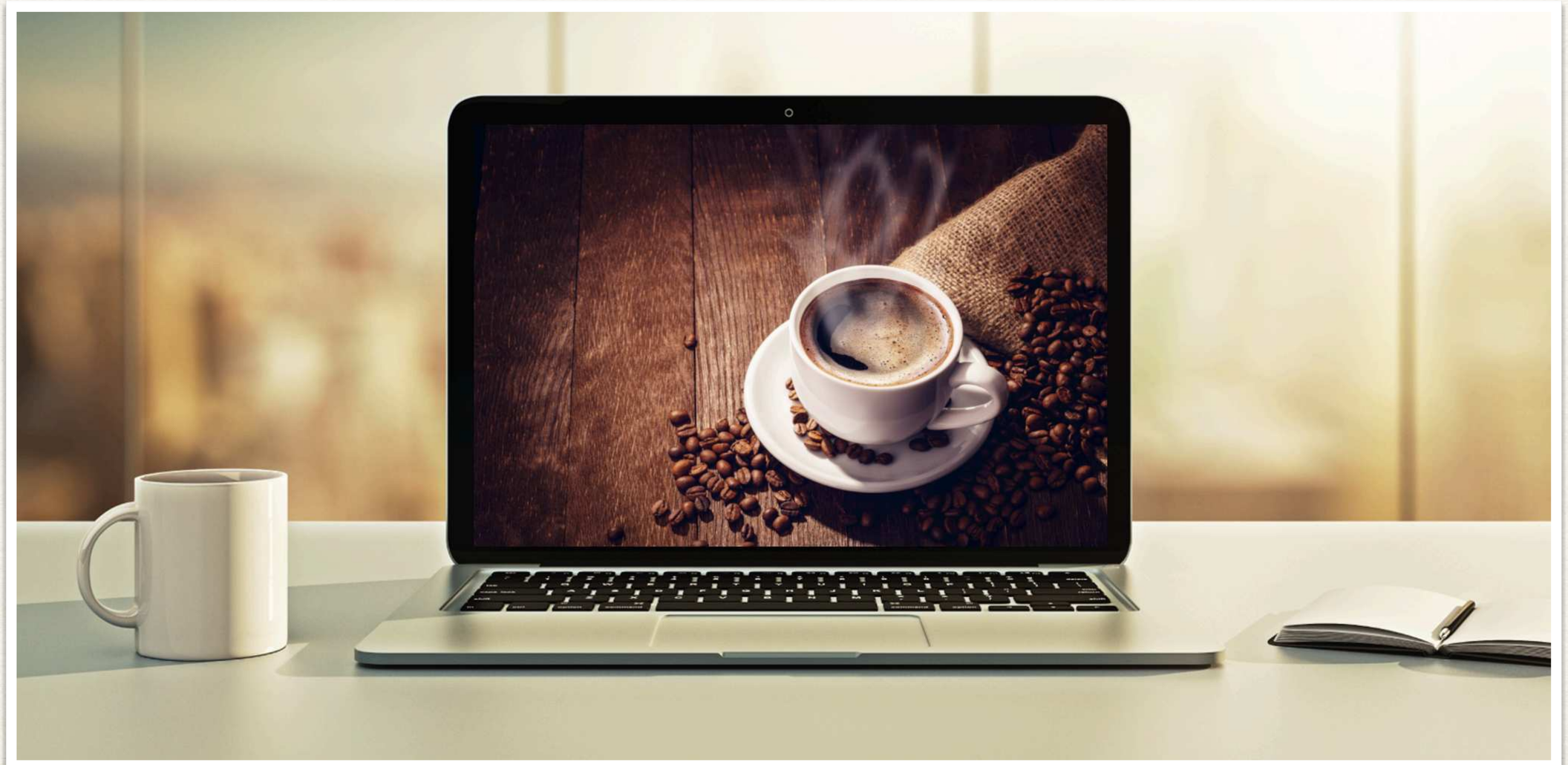
- As of now we need to remember to close our connection to our file to retrieve resources.

```
~~reader.close();~~
```

- We can have Java handle this for us using the Try with Resources syntax.

```
try (BufferedReader reader = new BufferedReader(new FileReader(story.txt))) {
   << code here >>
}
```

BufferedWriter to Create & Write



BufferedWriter

- Used to create and write a file to a disk drive
- Writes text to a character output stream
- Buffers characters for efficient writing of characters and strings

New BufferedWriter Object

New FileWriter Object

```
BufferedWriter writer = new BufferedWriter(new FileWriter(file.txt));
```

Creates or overrides

BufferedWriter

Wrap the BufferedWriter in a Try / Catch

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

  // Try with resources
  try (BufferedWriter writer = new BufferedWriter(new FileWriter(file.txt))) {

   // Write a new line on file.txt
      writer.write("Hello World!");

   // Create a new line on file.txt
      writer.newLine();

   // Write on the new line
      writer.write("This is a simple text file.");

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

Read our new file with a BufferedReader

Wrap the BufferedReader in a Try / Catch

```
try (BufferedReader reader = new BufferedReader(new FileReader(file.txt))) {

    String line;

 // Fetches a line at a time
    while ((line = reader.readLine()) != null) {
      System.out.println(line);
 }
}
catch(FileNotFoundException e) {
    e.printStackTrace();
}
```