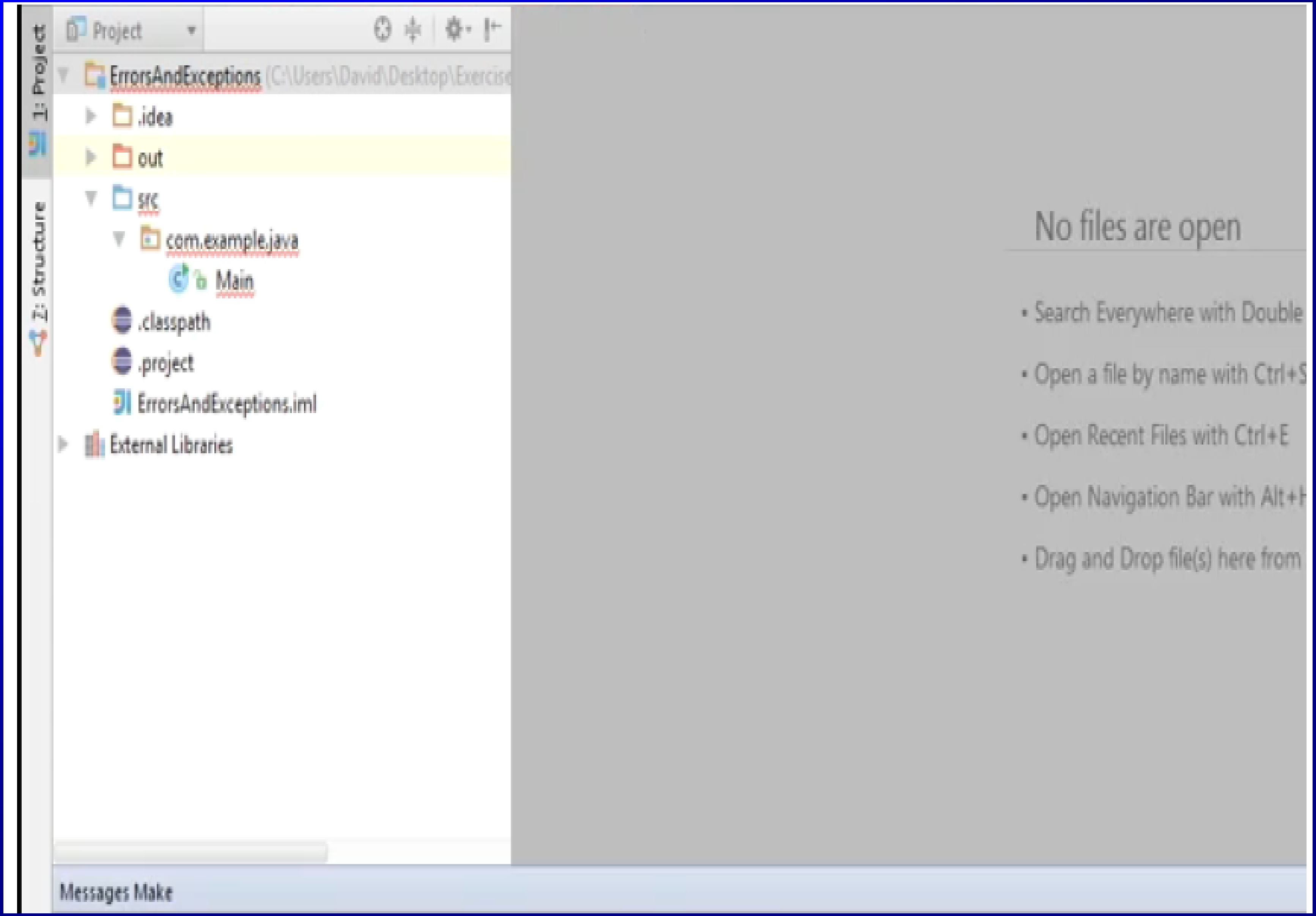
Exception Handling

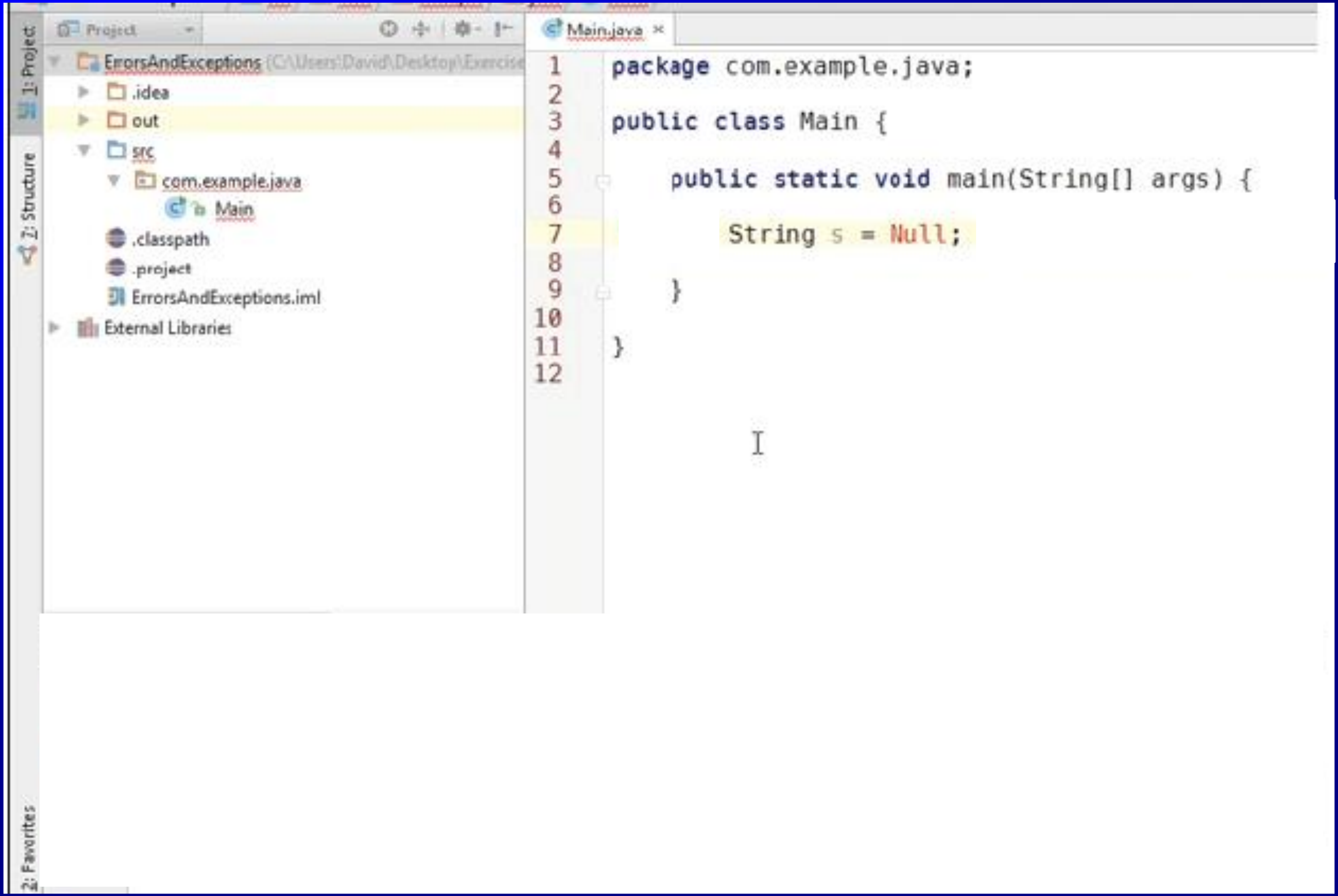
Motivations

When a program runs into a runtime error, the program terminates abnormally. How can you handle the runtime error so that the program can continue to run or terminate gracefully? This is the subject we will introduce in this chapter.



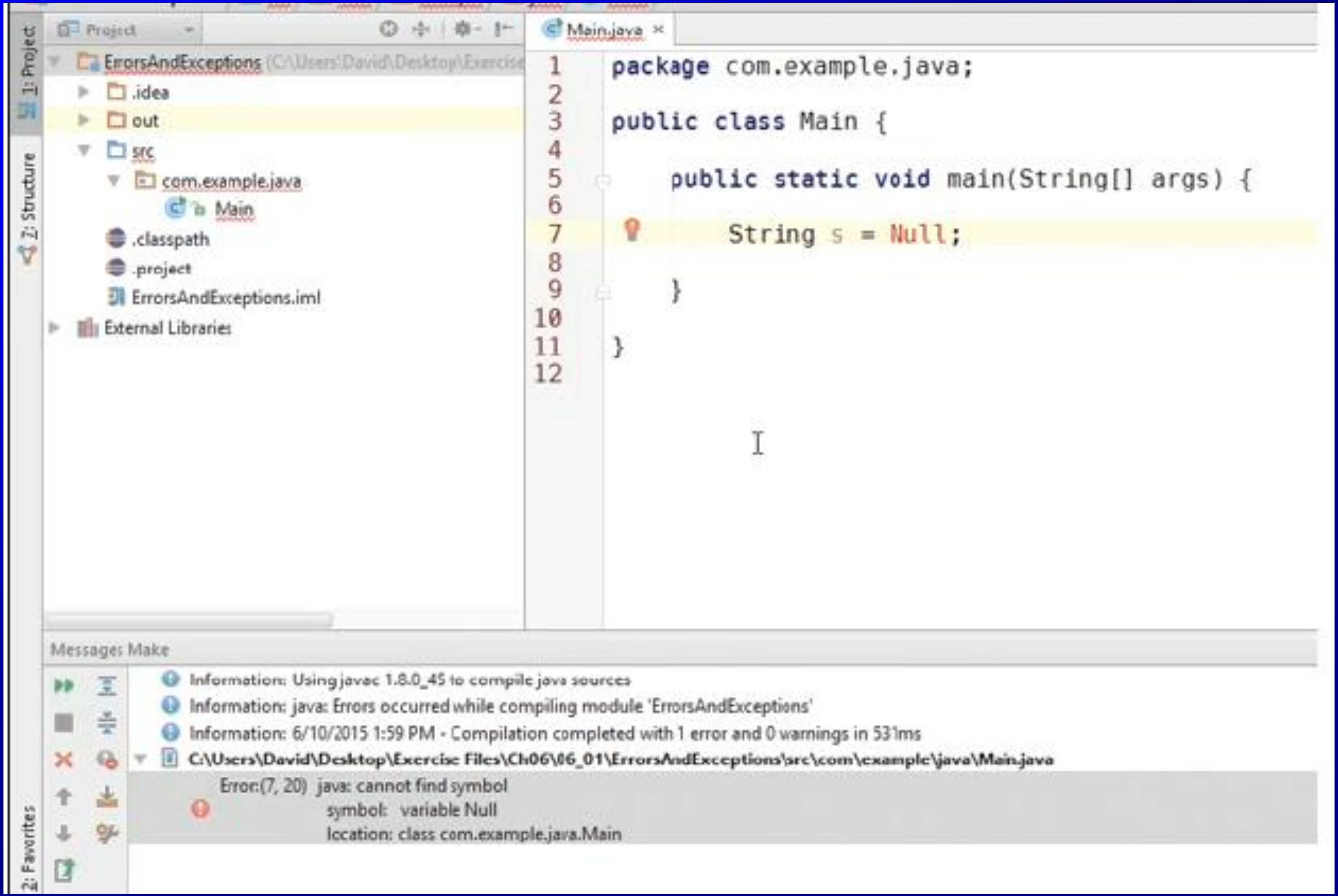
No files are open

- Search Everywhere with Double
- Open a file by name with Ctrl+S
- Open Recent Files with Ctrl+E
- Open Navigation Bar with Alt+H
- Drag and Drop file(s) here from



```
package com.example.java;

public class Main {

    public static void main(String[] args) {

        String s = Null;

    }

}
```

Types of errors



Project

ErrorsAndExceptions (C:\Users\David\Desktop\Exercise

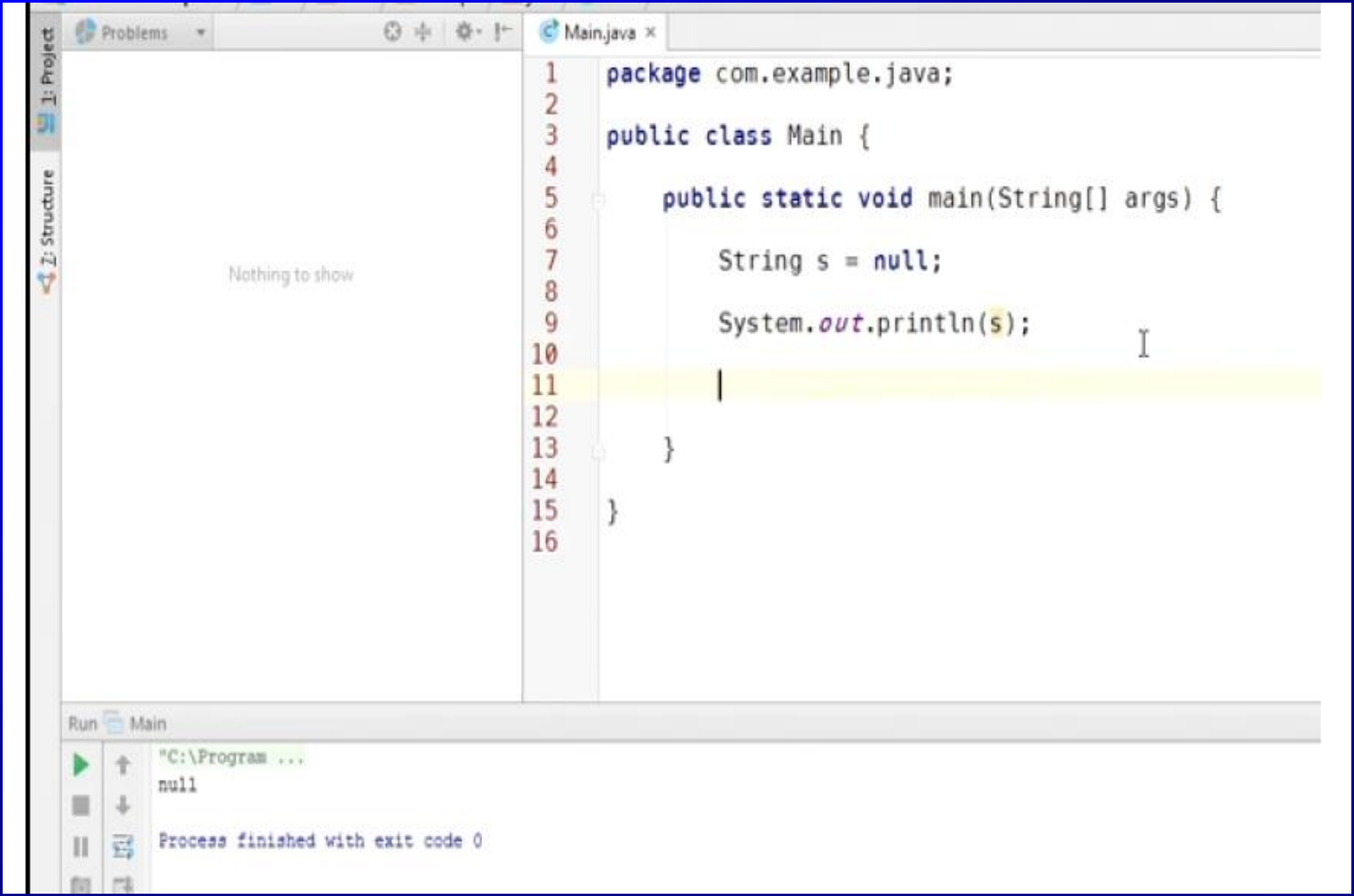
.idea
out
src
com.example.java
Main
.classpath
.project
ErrorsAndExceptions.iml
External Libraries

Main.java

```java
package com.example.java;

public class Main {

    public static void main(String[] args) {

        String s = Null;

    }

}
```

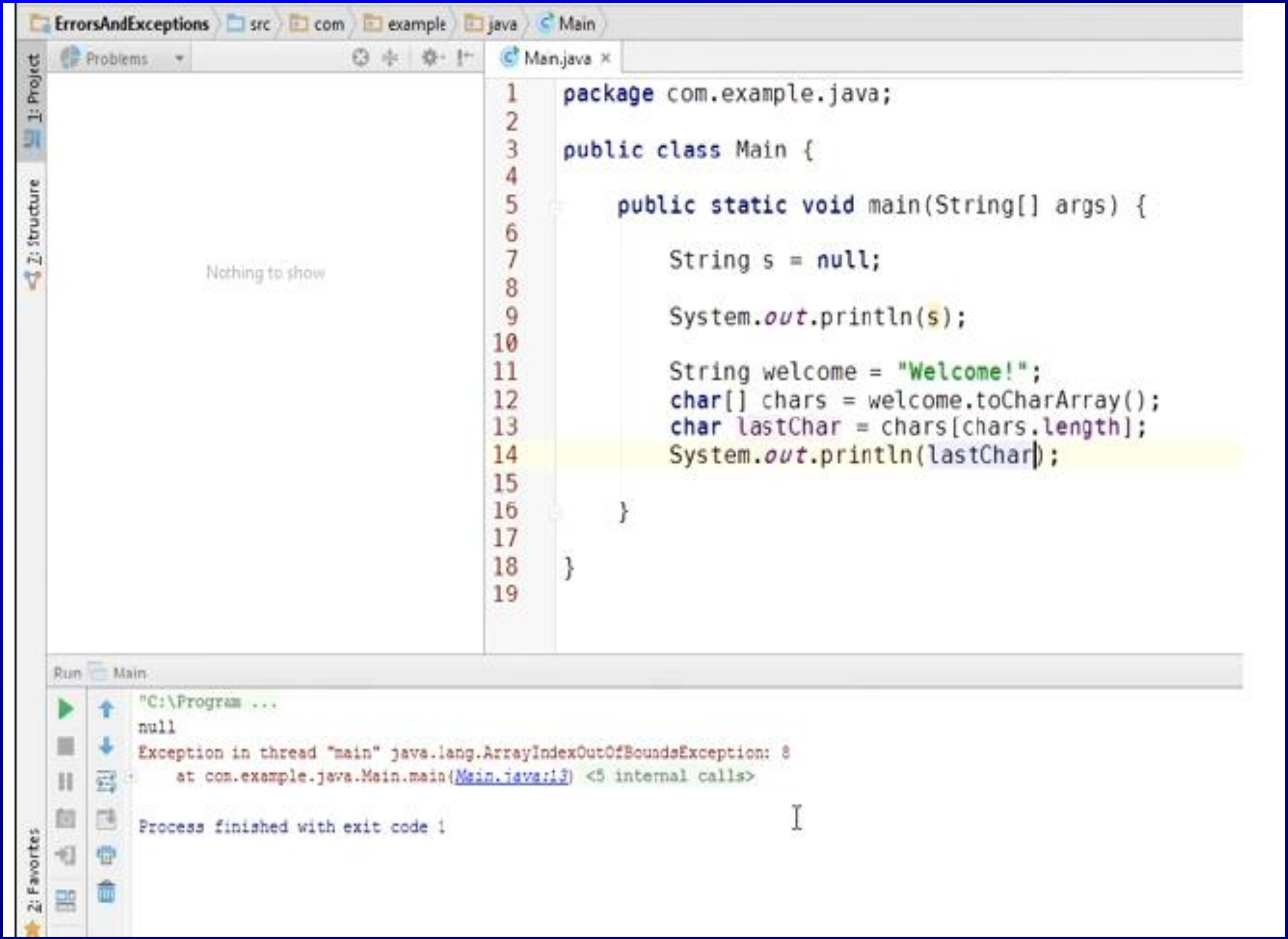
Messages Make

```
Information: Using javac 1.8.0_45 to compile java sources
Information: java: Errors occurred while compiling module 'ErrorsAndExceptions'
Information: 6/10/2015 1:59 PM - Compilation completed with 1 error and 0 warnings in 531ms
C:\Users\David\Desktop\Exercise Files\Ch06\06_01\ErrorsAndExceptions\src\com\example\java\Main.java
    Error:(7, 20)  java: cannot find symbol
        symbol:   variable Null
        location: class com.example.java.Main
```



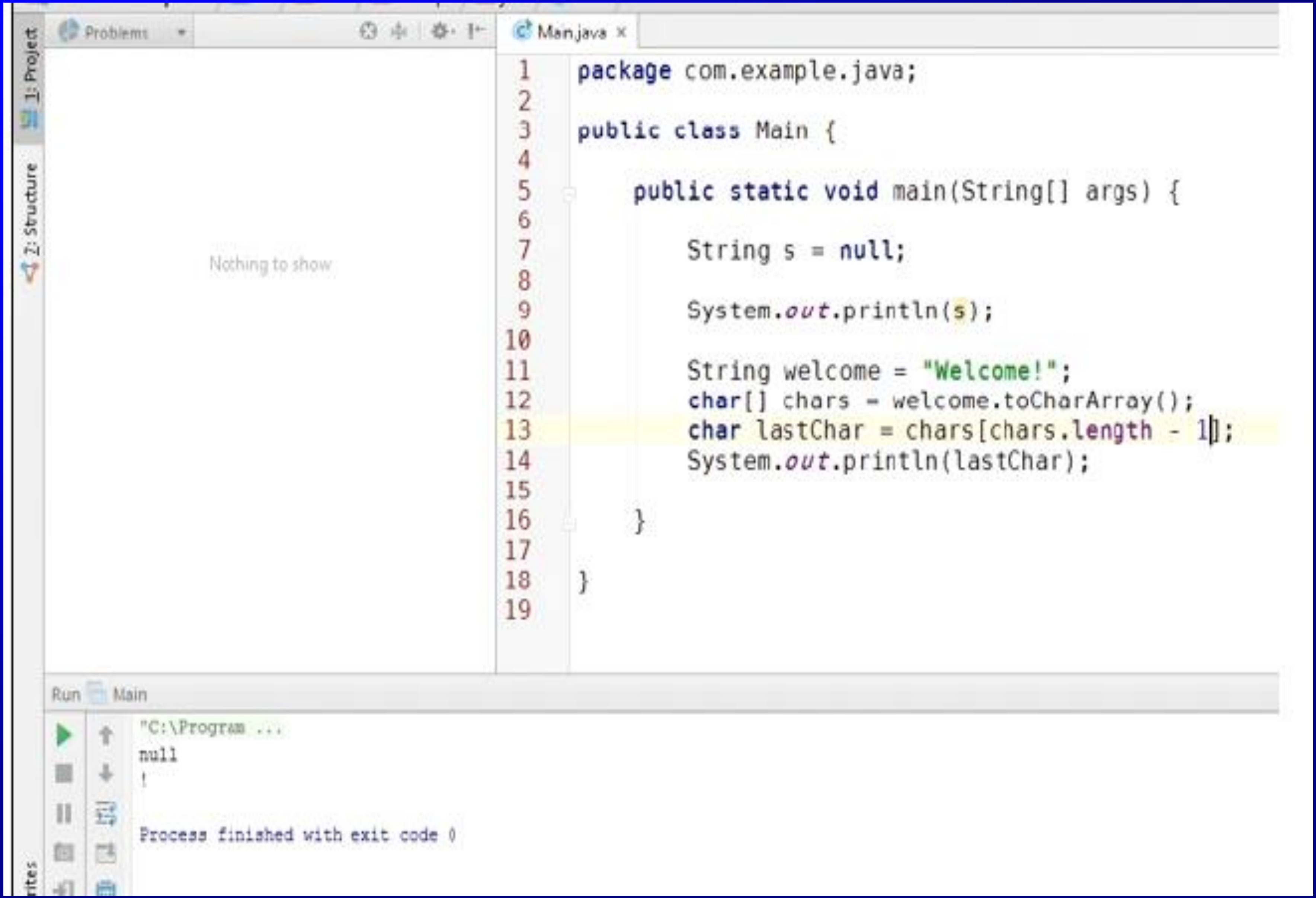
```
package com.example.java;

public class Main {

    public static void main(String[] args) {

        String s = null;

        System.out.println(s);



    }

}
```

```
"C:\Program ...
null

Process finished with exit code 0
```



```java
package com.example.java;

public class Main {

    public static void main(String[] args) {

        String s = null;

        System.out.println(s);

        String welcome = "Welcome!";
        char[] chars = welcome.toCharArray();
        char lastChar = chars[chars.length];
        System.out.println(lastChar);

    }

}
```

```
"C:\Program ...
null
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 8
    at com.example.java.Main.main(Main.java:13) <5 internal calls>

Process finished with exit code 1
```



```java
package com.example.java;

public class Main {

    public static void main(String[] args) {

        String s = null;

        System.out.println(s);

        String welcome = "Welcome!";
        char[] chars = welcome.toCharArray();
        char lastChar = chars[chars.length - 1];
        System.out.println(lastChar);

    }

}
```

```
"C:\Program ...
null
!

Process finished with exit code 0
```

Java Exception

- An exception (or exceptional event) is a problem that arises during the execution of a program. When an **Exception** occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

- An exception can occur for many different reasons. Following are some scenarios where an exception occurs.
- A user has entered an invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

- Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.
- Based on these, we have three categories of Exceptions. You need to understand them to know how exception handling works in Java.

- ***Checked exceptions*** − A checked exception is an exception that occurs at **the compile time**, these are also called as **compile time exceptions.** These exceptions cannot simply be ignored at the time of compilation, the programmer should take care of (handle) these exceptions.
- For example, if you use **FileReader** class in your program to read data from a file, if the file specified in its constructor doesn't exist, then a*FileNotFoundException* occurs, and the compiler prompts the programmer to handle the exception.

Example

```
import java.io.File;
import java.io.FileReader;
public class FilenotFound_Demo {
   public static void main(String args[]) {
      File file = new File("E://file.txt");
      FileReader fr = new FileReader(file);
   }
}
```

Note – Since the methods **read()** and **close()** of FileReader class throws IOException,

you can observe that the compiler notifies to handle IOException, along with FileNotFoundException

If you try to compile the above program, you will get the following exceptions.

Output :

```
C:\>javac FilenotFound_Demo.java
FilenotFound_Demo.java:8: error: unreported exception FileNotFoundException; must be caught or
                                                                             declared to be thrown
        FileReader fr = new FileReader(file);
                    ^1 error
```

Difference between error and exception

Errors indicate serious problems and abnormal conditions that most applications should not try to handle. Error defines problems that are not expected to be caught under normal circumstances by our program. For example memory error, hardware error, JVM error etc.

Exceptions are conditions within the code. A developer can handle such conditions and take necessary corrective actions.

- Few examples –
- DivideByZero exception
- NullPointerException
- ArithmeticException
- ArrayIndexOutOfBoundsException

Exception-Handling Overview

Show runtime error

Quotient

Run

Fix it using an if statement

QuotientWithIf

Run

With a method

QuotientWithMethod

Run

Exception Advantages

QuotientWithException

Run

Now you see the *advantages* of using exception handling. It enables a method to throw an exception to its caller. Without this capability, a method must handle the exception or terminate the program.

Handling InputMismatchException

InputMismatchExceptionDemo

Run

By handling InputMismatchException, your program will continuously read an input until it is correct.

Advantages of Exception Handling

- Exception handling allows us to control the normal flow of the program by using exception handling in program.
- It throws an exception whenever a calling method encounters an error providing that the calling method takes care of that error.
- It also gives us the scope of organizing and differentiating between different error types using a separate block of codes. This is done with the help of try-catch blocks.

When to Use Exceptions

When should you use the try-catch block in the code? You should use it to deal with unexpected error conditions. Do not use it to deal with simple, expected situations. For example, the following code

```
try {
  System.out.println(refVar.toString());
}
catch (NullPointerException ex) {
  System.out.println("refVar is null");
}
```

Declaring Exceptions

Every method must state the types of checked exceptions it might throw. This is known as *declaring exceptions*.

```
public void myMethod()
  throws IOException

public void myMethod()
  throws IOException, OtherException
```

When to Throw Exceptions

- An exception occurs in a method. If you want the exception to be processed by its caller, you should create an exception object and throw it. If you can handle the exception in the method where it occurs, there is no need to throw it.

Throwing Exceptions

When the program detects an error, the program can create an instance of an appropriate exception type and throw it. This is known as *throwing an exception*. Here is an example,

```
throw new TheException();
```

```
TheException ex = new TheException();
throw ex;
```

Throwing Exceptions Example

```
    /** Set a new radius */
public void setRadius(double newRadius)
        throws IllegalArgumentException {
    if (newRadius >= 0)
        radius =  newRadius;
    else
        throw new IllegalArgumentException(
          "Radius cannot be negative");
}
```

Try Catch in Java – Exception handling

One of the main components of exception handling is the **try-catch** block. This is a special code structure that works in the following way:

```
try {
    // code goes here that might
    // throw an exception
    // i.e.  "try this code"
} catch (Exception ex) {
    // code goes here that handles
    // the exception
    // you could display a message,
    // exit the program, etc.
    // i.e. "if the code I tried throws
    // an exception, catch it here and
    // do whatever"
}
```

Try Catch in Java – Exception handling

What is Try Block?

The try block contains a block of program statements within which an exception might occur. A try block is always followed by a catch block, which handles the exception that occurs in associated try block. A try block must followed by a Catch block or Finally block or both.

Syntax of try block

```
try{
        //statements that may cause an exception
  }
```

What is Catch Block?

A catch block must be associated with a try block. The corresponding catch block executes if an exception of a particular type occurs within the try block. For example if an **arithmetic exception** occurs in try block then the statements enclosed in catch block for arithmetic exception executes.

Syntax of try catch

```
try {
      //statements that may cause an exception
  }
catch (exception(type) e(object))
 {
      //error handling code
  }
```

One problem we've noticed a lot when working with user inputs is that if the user types in an invalid value, the program crashes. For example, try the following program:

```
import java.util.*;
 public class TestExceptions {
     public static void main(String[] args) {
         Scanner in = new Scanner(System.in);
         System.out.print("Enter a number: ");
        int number = in.nextInt();
         System.out.println("Root: " + Math.sqrt(number));
    }
}
```

As long as the user types any number, the program works fine. But what if they type a string value? Try your program and enter the value "two" at the prompt. You should see the following errors:

```
Enter a number: two
Exception in thread "main"
java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:840)
        at java.util.Scanner.next(Scanner.java:1461)
        at java.util.Scanner.nextInt(Scanner.java:2091)
        at java.util.Scanner.nextInt(Scanner.java:2050)
        at TestExceptions.main(TestExceptions.java:10)
```

This is a **stack trace** and it's telling you what exception occurred and where. In this example, the first line is saying that an exception occured in the main() method, and that this exception was an "InputMismatchException". At the bottom of the stack trace, we can see that the problem started on line 10 of the main() method. In my code, that's the line

```
        int number = in.nextInt();
```

On the next line up in the stack trace, it indicates that the Scanner.nextInt() method generated the error. All of the nextX() methods (e.g. nextInt(), nextDouble(), etc.) will throw this exception if they receive a value that a valid numeric.

For example, only digits and the decimal point are acceptable for doubles and only digits are acceptable for ints.

- This means that if the user types letters or symbols in our program, an InputMismatchException is thrown.
- We can't assume that are users will always type valid input all of the time. Sometimes users forget to read instructions, sometimes they forget what the instructions are, and sometimes they're just daft! We must never assume that user input will always be correct, so a good program always uses various techniques to ensure that invalid inputs don't crash a program. One of these techniques is **exception handling**.

Modify the example so it uses a try-catch block:

```
import java.util.*;
public class InputErrors {
    public static void main(String[] args) {
        try {
            Scanner in = new Scanner(System.in);
            System.out.print("Enter a number: ");
            int number = in.nextInt();
            System.out.println("Root: " + Math.sqrt(number));

        } catch (Exception ex) {
            System.out.println(ex.getMessage());
        }
    }
}
```

When you re-compile and run you're program, you'll notice that you don't get the error message on the console. Instead, you simply see:

```
null
```

In this case, the error message inside this Exception object is not very helpful. The value "null" is used when an object contains nothing at all, so seeing "null" here means that the Exception object's message (String object) is empty. Change the println() in the catch-block to print the entire exception object, instead:

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

This time when you run the program you see:

```
java.util.InputMismatchException
```

This is the fully qualified name of the Exception object (that means that you see the name of the class, preceeded by its package name). Again, this is not too helpful, especially to a non-technical person using your program!

Now edit the same line, but this time print a nice, friendly error message that would make more sense to the user:

```
System.out.println("Error: Not a whole number. Exiting..");
```

When you re-compile and run your program, you'll notice that you don't get the error messages on the console. Instead, you get a message with Error: Not a whole number. Exiting...

Important!

Never leave the catch-block empty!! If you do this, and there are other exceptions happening in your code that you didn't think about, you'll never find out about them because they'll be caught by your catch-block:

since it's empty, nothing will show on the screen; you'll receive no indication at all that an exception has occurred!

Always make sure you have something in your catch block, even if it's just **ex.printStackTrace();** until you get around to creating some nicer error messages.

Important!

A try-block must be accompanied by a catch-block or a finally-block. You'll learn more about finally-blocks in term 2. A try-block can have multiple catch-blocks, which you'll also learn about in term 2.

Catching Exceptions -

```
try {
  statements;  // Statements that may throw exceptions
}
catch (Exception1 exVar1) {
  handler for exception1;
}
catch (Exception2 exVar2) {
  handler for exception2;
}
...
catch (ExceptionN exVar3) {
  handler for exceptionN;
}
```

An example of Try catch

```
class Example1 {
  public static void main(String args[]) {
    int num1, num2;
    try {
      // Try block to handle code that may cause exception
      num1 = 0;
      num2 = 62 / num1;
      System.out.println("Try block message");
    } catch (ArithmeticException e) {
        // This block is to catch divide-by-zero error
        System.out.println("Error: Don't divide a number by zero");
      }
    System.out.println("I'm out of try-catch block in Java.");
  }
}
```

Output:

```
Error: Don't divide a number by zero
I'm out of try-catch block in Java.
```