2021/2022(1)
IF184301 Object Oriented Programming

Lecture #4b

Exception and I/O

Misbakhul Munir IRFAN SUBAKTI
司馬伊凡
Мисбакхул Мунир Ирфан Субакти

All about exceptions: Errors happen

- One of the problems with programming is users!
- We never know when errors will occur: the best we can do is guess what might cause them, and try to mitigate their effects.
- Sometimes, in fact, errors can occur because of the programmer not anticipating what problems could arise.
- In order to cover all bases, Java provides us with a convenient way to tell us when something has gone wrong: `Exceptions` and `Errors`.
- The word `Error` actually has a special (bad) meaning in Java, and we won't cover it further. It suffices to say that if our code produces an `Error` and not an `Exception`, something has gone wrong enough that Java cannot recover.
- Whenever something goes wrong in Java, it produces typically an *exception*. We will probably have seen a few of these already: `ArrayIndexOutOfBoundsExceptions`, `InputMismatchExceptions`, `NullPointer Exceptions` to name some of the most common. They're Java's indication that something is wrong, but that we *could* do something to fix it. Have a look at the code below: what error could occur with it?

Errors happen: Example

```
import java.io.*;
import java.util.*;
public class InputTest {
     public static void main(String[] args) {
          Scanner s = new Scanner(System.in);
          ArrayList<Integer> intList = new ArrayList<Integer>();
          System.out.println("Enter some numbers!");
          int currentVal = s.nextInt();
          while (currentVal != 99) {
               System.out.println("Adding " + currentVal);
               intList.add(currentVal);
               currentVal = s.nextInt();
          }
          System.out.println("Exiting: List is " + intList.toString());
     }
}
```

- If we noticed that the possible exception is to do with the user's input, well done. Consider what the line `int currentVal = s.nextInt()` expects: the user should enter a number. If the user enters a `String` that cannot be interpreted as a number such as "`b`", Java won't be happy:

Errors happen: Example (continued)

```
D:\OOP>java InputTest
Enter some numbers!
1
Adding 1
2
Adding 2
b
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 InputTest.main(InputTest.java:14)
```

- Ouch! What does this mean? Well, it means we need to fix something.
- What do we need to fix? That's usually indicated by the first line of the exception. We can safely ignore the `in thread "main"` bit, and look at the exception type, `java.util.InputMismatchException`. This makes sense: `nextInt` expects an `int`, and we entered a `String` that cannot be interpreted as an `int`. What's then printed is called a `stack trace`: this is all of the methods that were called up to the point where the exception occurred, beginning with the most recent. At the bottom, we can see that our class, `InputTest.java`, caused the error at line 14, which is the second occurrence of `s.nextInt()`, in the loop.

Errors happen: Example (continued)

- Let's have a look at another example, which we'll put after that code:

```
Scanner s = new Scanner(System.in);
String[] stringArray = new String[5]; // Only five elements...
System.out.println("Enter five strings");
for (int i = 1; i <= 5; i++) {
      String line = s.nextLine();
      stringArray[i] = line;
      System.out.println("Added " + line + " at index " + i);
}
System.out.println("Done: " + Arrays.toString(stringArray));
```

- What error will occur here? This time, it's to do with the *length* of the array:

```
Enter five strings
this
Added this at index 1
is
Added is at index 2
a
Added a at index 3
string
Added string at index 4
and
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
at InputTest.main(InputTest.java:22)
```

Errors happen: Example (continued)

- The error is much easier to understand here: Again, we're told we have an exception in the main thread (which we can ignore).
- Then we can see that it's an `ArrayIndexOutOfBoundsException`, and the number 5, which relates to the *index of the array* at which the exception occurred.
- This makes sense, because the array is size 5 (meaning the index of the last element is 4), and we're trying to access index 5. As expected, line 22 points to the line `stringArray[i] = line`.

A little more about exceptions

- There are hundreds, if not thousands of types of Exception. When we see an exception on our screen, it’s because it's been *thrown* by Java. As such, all Exceptions extend (directly or otherwise) the class `Throwable`: Look at `ArrayIndexOutOfBoundsException` in the API:

```
java.lang.Object
   extended by java.lang.Throwable
      extended by java.lang.Exception
         extended by java.lang.RuntimeException
            extended by java.lang.IndexOutOfBoundsException
               extended by java.lang.ArrayIndexOutOfBoundsException
```

- Later, we'll consider writing our own exceptions, which will also extend the `Exception` (or `RuntimeException` class).
- That actually raises an interesting point—there are two sorts of Exception: those that can occur anywhere, and as such can't really be planned for by the compiler, and those that Java *knows* can occur, and as such forces us to do something about.

A little more about exceptions

- Exceptions like `ArrayIndexOutOfBoundsException` can occur anywhere—since they happen during the execution of a program, without predictability, they are called *runtime exceptions*, and extend `RuntimeException` somewhere in their hierarchy.
- Other exceptions just extend the `Exception` class, without `RuntimeException`, somewhere in their hierarchy. In this case, Java knows that it's possible the exception could occur, and forces us to do something about it. We'll talk more about this type of exception later.

Do something about it

- Well, we've said that exceptions can be *thrown*. To handle them, they need to be *caught*. To do this, we wrap the code that could cause an exception in something called a `try...catch` block:

```
try {
      // Code that could cause an exception
} catch (SomeExceptionType et) {
      // What to do when that exception occurs
}
```

- Our main aim in catching exceptions is to *prevent the program quitting* with an error. So, we surround the '*dangerous*' code with a '`try` block' (the `try { ... }` code). If no exceptions occur, we continue with execution of the code below the block. If an exception *does* occur, we consider the `catch` block(s). If the exception thrown matches the type in the round brackets, we give it a name, `et` in the example above, and then do something with it. We could have multiple `catch` blocks: if it's not the first exception, try the next down, and so on:

```
try {
      // Code that could cause an exception
} catch (SomeExceptionType et) {
      // What to do when that exception occurs
} catch (AnotherExceptionType et2) {
      // Do something else
} catch (Exception e) {
      // This will catch all other exceptions
}
```

- We should think of these multiple `catch` blocks as being progressively bigger nets, each catching a wider range of exceptions, with the first one being the most specific.

What goes in the `catch` block?

- Let's look at an example like the one above, and then think about it:

```
Scanner s = new Scanner(System.in);
ArrayList<Integer> intList = new ArrayList<Integer>();
System.out.println("Enter some numbers!");
int currentVal;
while (true) {
     try {
          currentVal = s.nextInt();
          if (currentVal == 99) {
               break;
          }
          System.out.println("Adding " + currentVal);
          intList.add(currentVal);
     } catch (InputMismatchException ime) {
          System.out.println(ime + ": please enter only numbers.");
          s = new Scanner(System.in);
     }
}
```

- Now, every time we enter something that isn't a number, what happens? Java throws an exception, and it's caught by the `catch` block. We print out the type of the exception, reset the Scanner to how it was initialised, and then finish the code block (which takes us back to the top of the while loop). All without crashing!

What goes in the `catch` block? (continued)

- Sometimes, we'll want to do something else when we discover an error. As can be seen in example aside, if the number for a given grayscale value exceeds 255, Java will raise an `IllegalArgumentException` (very common where parameters should be within certain bounds). We could use the `catch` block in this case to reset the value to 255, to avoid halting the program halfway through. Often, we might just print out that an error has occurred, and allow the user to continue below the try block.

```
1  public class IllegalArgumentExceptionExample {
2      public static boolean isExceeded(int x) {
3          boolean result = false;
4          if (x > 255) {
5              throw new IllegalArgumentException("Value exceeds 255");
6          } else {
7              result = true;
8          }
9          return result;
10     }
11     public static void main(String[] args) {
12         int value = 270;
13         try {
14             // Assume, we deal with Grayscale value: 0-255
15             // Check the value's range
16             if (isExceeded(value)) {
17                 System.out.println(value + " is a valid Grayscale value");
18             }
19         } catch (IllegalArgumentException ie) {
20             System.out.println(value + " is an invalid Grayscale value");
21             System.out.println("Resetting the value to 255");
22             value = 255;
23         }
24         System.out.println("Continue processing the value...");
25     }
26 }
```

I/O: Scanner

- By now, we all know how to make Java print, using the `System.out.println` command. Quite often though, we need also to be able to read in:

```
Scanner sc = new Scanner(System.in);
String aLineOfText = sc.nextLine();
```

- Scanner is a useful tool that allows us to grab input from the keyboard. In order to use it, we need to add `import java.util.Scanner` to the top of our classes, to tell Java to include it in our code. In this example, when we write `sc.nextLine()`, we're reading a line of text from the keyboard, and assigning it to `aLineOfText`. Scanner provides a number of other useful input types, such as `sc.nextInt()` or `sc.nextDouble()`:

```
System.out.println("Pick an integer:");
int value = sc.nextInt();
System.out.println("You picked " + value +
    ". The square of that is " + (value * value) + ".");
```

- If we enter something that isn't an `int`, Java will throw an error and crash. In these cases an exception will be thrown.

More powerful input

- Much input in java is done with streams, which are read by stream readers. An example is keyboard input: this can occur via the `InputStream` object `System.in`:

```
InputStreamReader isr = new InputStreamReader(System.in);
char character = (char) isr.read(); // Read a single character
```

- As we might imagine, reading a single character at a time is a little tiresome. What is needed is some way to *buffer* characters into a line at a time, and for this we can use a `BufferedReader`, which *wraps around* an `InputStreamReader`:

```
import java.io.*;
public class MyReader {
    public MyReader() {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
```

More powerful input (continued)

- Now, we can be quite specific about how we want to read from the keyboard. Let's say that we want to read lines from the keyboard into an ArrayList of Strings until the user enters "stop":

```
ArrayList<String> stringList = new ArrayList<String>();
String currentLine = "";
while (!(currentLine = br.readLine()).equals("stop")) {
    stringList.add(currentLine);
}
// Print the list out
```

- Of course, we could simplify the code about to make it more comprehensible:

```
String currentLine = br.readLine();
while (!currentLine.equals("stop")) {
    stringList.add(currentLine);
    currentLine = br.readLine();
}
```

More powerful input (continued)

- One particularly useful thing about using `BufferedReaders` with `InputStreamReaders`, though, is that they can be applied to *anything* which provides an `InputStream` to read from: in the following we could read from a webpage by specifying a URL:

```
BufferedReader webReader =
    new BufferedReader(
        new InputStreamReader(
            new URL("http://www.myURL.com").openStream()));
```

- We could also use `BufferedReader` to read directly from a file. We do this by using the class `FileReader`, which extends `InputStreamReader` directly, and takes a `File`, or a filename, as an argument:

```
BufferedReader fr =
    new BufferedReader(
        new FileReader(
            new File("/path/to/my/filename.txt")));
// Equivalent to
BufferedReader fr2 =
    new BufferedReader(
            new FileReader("/path/to/my/filename.txt"));
```

- Which one we should use? `Scanner` or `BufferedReader`?