2022/2023(1)
IF184301 Object Oriented Programming

Lecture #4c

Exception: More about

Misbakhul Munir IRFAN SUBAKTI
司馬伊凡
Мисбакхул Мунир Ирфан Субакти

More about exception

- Pretty much everything in the package `java.io` has the potential to generate *exceptions*, which we need to handle.
- We should be able to see the obvious potential for exceptions when trying to read from a file. What if the file isn't there? Java obviously can't read from it, and so has to raise an exception.
- However, the sort of exceptions we're dealing with here aren't the `RuntimeExceptions` from above. They’re known as *checked* exceptions.

Exception: Unchecked vs checked

- We've mentioned a number of exceptions already. Remember them?

```
String[] arr = new String[6];
System.out.println(arr[6]); // What happens here?
System.out.println(arr[3].length) ; // What about here?
Object x = new Integer(0);
System.out.println((String) x); // How about now?
```

- All of the above examples are so-called *unchecked* exceptions. There are a couple of reasons for this name.
- Unchecked exceptions all extend the class `RuntimeException`, and the compiler will *not force us to catch them*. This is because it cannot usually tell whether these exceptions are likely to occur during runtime, and *that* is because unchecked exceptions are generally down to programmer error, i.e., the programmer is probably doing something *stupid*. That much should be clear from the above examples.
- Generally, if we get an ***unchecked exception*** from our code, it's ***our fault***, and we should do something to try to fix it. Simply catching these exceptions may not be getting to the root of the problem.

Exception: Unchecked vs checked (continued)

- *Checked* exceptions are slightly different. Java *knows* that these are likely to occur, and *forces* us to catch them. The `java.io` package is a good example. If we try to compile the following code:

```
import java.io.*;
public class Test {
    public static void main(String[] args) {
        BufferedReader br = new BufferedReader(new
            FileReader("filename.txt"));
    }
}
```

- compilation wouldn't work. The compiler will tell us:

```
Test.java:4: unreported exception java.io.FileNotFoundException;
    must be caught or declared to be thrown
```

Exception: Unchecked vs checked (continued)

- The semantics of this sentence need to be understood. Either the exception `FileNotFoundException` must be caught by our code, or our code must say that it deliberately *doesn't* handle the exception, but *throws* it to the method from where it was called. In the code above, it would be a very bad idea to allow `main` to declare that it throws `FileNotFoundException`. Instead, we should handle the possibility that the file isn't there:

```
public static void main(String[] args) {
    try {
        BufferedReader br = new BufferedReader(new FileReader("filename.txt"));
    } catch (FileNotFoundException fnfe) {
        System.out.println("File not found: " + fnfe.getMessage());
    } catch (IOException ioe) {
        System.out.println("Miraculously, some other IO exception: " +
                ioe.getMessage());
    }
    // Read from file here
}
```

- As we might have guessed, the above code is as such not useful (why not?), though it would now compile.

Causing exceptions to happen

- Sometimes, we want to force our own exceptions to happen, if a user does something which we want to forbid. For this reason, we can cause an exception with the `throw` keyword:

```
public void getSomeNumbers(int first, int second) {
      if (first + second >= 20) {
            throw new IllegalArgumentException("The numbers you've entered are not acceptable.");
      }
      System.out.println("Your numbers are fine. Well done." );
}
```

- A couple of important things to finally note. If we throw an exception, it must be caught somewhere (or the program crashes because of it). E.g.:

```
public void myExceptionHandlingMethod() {
      try {
            dangerousMethod();
      } catch (IdiotUserException iue) { /* handle error */ }
}
public void dangerousMethod(User u) throws IdiotUserException {
      if (u.isIdiot()) {
            throw new IdiotUserException();
      }
}
```

- Lastly, it's considered good practice to only throw exceptions when you want to show that *error* behaviour has occurred. Exceptions shouldn't be used for other reasons.

Finally

- When dealing with files (or I/O generally), there's an extra part to the `try...catch` block, called the `finally` block. What we put in this block is things that *must be done*, irrespective of whether an exception was thrown or not. The most common that we need to worry about is when reading or writing using streams. Those streams must be *closed* once you're done with them. In the example below, we're copying a file:

```
FileInputStream source = null;
FileOutputStream target = null;
try {
      source = new FileInputStream ("/path/to/my/filename.txt");
      target = new FileOutputStream("/path/to/copy.txt");
      for (int c = source.read(); c != -1; c = source.read()) {
            target.write(c);
      }
} catch (IOException e) {
      System.err.println("IO error " + args[0]);
      System.err.println(e.getMessage());
      exitCode = abnormalTermination; // We've changed our mind.
} finally {
      try {
            source.close();
            target.close();
      } catch(IOException e) { /* handle error */ }
}
```

- If we were just reading using a `BufferedReader`, we should call `close` on that. Note that as the `close` method itself can throw an `IOException`, it too should be enclosed in a `try...catch` (this could get complicated)!