Java Programming 2
Exceptions

Mary Ellen Foster

MaryEllen.Foster@glasgow.ac.uk

Image: “Serious orange juicing” by Leonid Mamchenkov
https://flic.kr/p/xXdD42

Exceptions

When an error occurs in program execution, an `Exception` is **thrown**

(Exceptions are also Java objects like any other; parent class is `java.lang.Exception`)

Unless the exception is **caught**, the entire program will crash

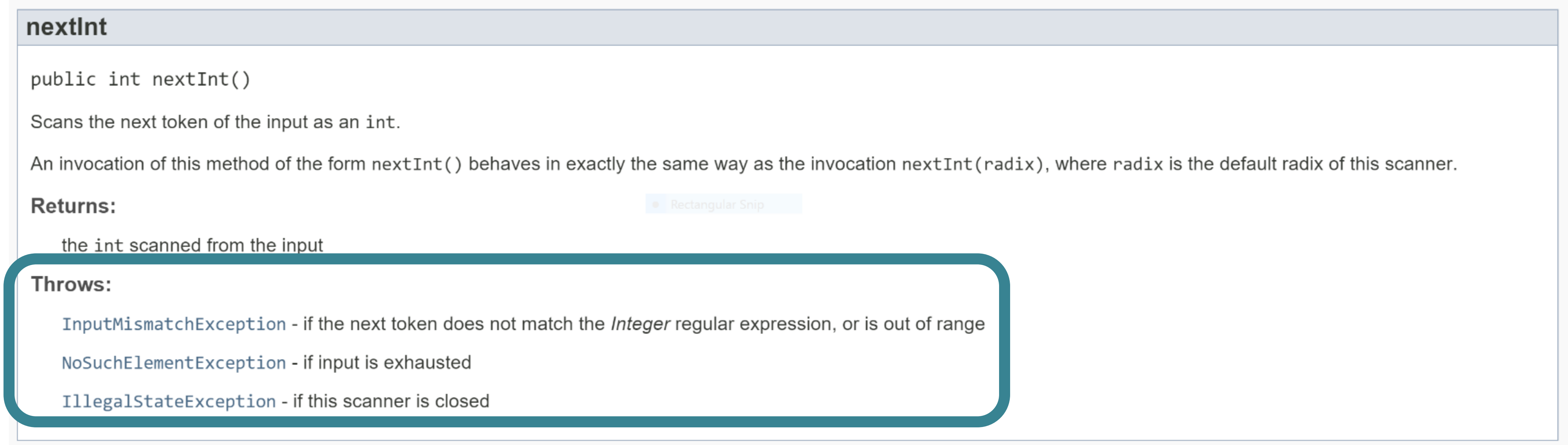
A sample Exception ...

```
Scanner s = new Scanner (System.in);
int n = s.nextInt();
```

- 1.5
- Abc
- 11111111111111111111

```
Exception in thread "main"
java.util.InputMismatchException
at java.util.Scanner.throwFor(Unknown Source)
at java.util.Scanner.next(Unknown Source)
at java.util.Scanner.nextInt(Unknown Source)
at java.util.Scanner.nextInt(Unknown Source)
at Test.main(Test.java:8)
```

Details of `Scanner.nextInt()`



nextInt

`public int nextInt()`

Scans the next token of the input as an `int`.

An invocation of this method of the form `nextInt()` behaves in exactly the same way as the invocation `nextInt(radix)`, where `radix` is the default radix of this scanner.

Returns:

the `int` scanned from the input

Throws:

`InputMismatchException` - if the next token does not match the *Integer* regular expression, or is out of range

`NoSuchElementException` - if input is exhausted

`IllegalStateException` - if this scanner is closed

http://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html#nextInt()

Java details

```
public class Scanner {
     // …
     public int nextInt()
          throws InputMismatchException,
          NoSuchElementException,
          IllegalStateException
     {
          // …
     }
}
```

Checked and unchecked exceptions

UNCHECKED EXCEPTIONS

- Do not need to be explicitly handled
  - Program will still compile and run without any special handling
- Generally indicate **programming/logic bugs** that an application cannot reasonably recover from
- Example: `ArrayIndexOutOfBoundsException`

CHECKED EXCEPTIONS

- Must be explicitly handled
  - Program will not compile unless you deal with them somehow
- Generally indicate conditions that a well-written application should anticipate and recover from
- Example: `FileNotFoundException`

More on checked/unchecked

"Unchecked Exceptions – The Controversy"
https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html

If a method specifies a checked exception, that is part of the method's public interface – anyone who calls that method should deal with exceptional cases

Why not just make everything checked?

- Runtime (unchecked) exceptions represent **programming problems**
- They can occur **anywhere** in a program and can be **numerous**
  - *e.g., in theory, every time you do anything on any object it could throw a NullPointerException*

Why not just make everything unchecked and not worry about try/catch?

- Client code should be prepared to deal with "expected" exceptional cases (file not found, device not turned on, …)

Handling exceptions #1: Catching

Wrap a `try {}` block around any code that might throw an Exception

Must be followed by one (or more) `catch {}` blocks

- First one whose parameter matches the thrown exception is executed

Optional `finally {}` block

- Executed after entire rest of the try block

```
try {
        // code that might
        // throw Exception
} catch (Exception ex) {
        // deal with it
} finally {
        // clean up
}
```

Handling exceptions #2: Passing on

If you do something that might throw an exception, you can add that exception to the `throws` clause of the current method

Then anyone who calls your method will need to handle the exception (by catching or passing on)

```
public void doSomething()
        throws IOException
{
        // code that might
        // throw IOException
}
```

Getting the details of an Exception

Every Exception has

- A **message** (`Exception.getMessage()`)
- A **call stack** – the sequence of method calls that ultimately resulted in the error

If you use `ex.printStackTrace()` inside a handler, it will print the stack trace

- Often has line numbers, at least in your own code
- Helpful for debugging!

```
Exception in thread "main"
java.util.InputMismatchException
 at java.util.Scanner.throwFor(Unknown
Source)
 at java.util.Scanner.next(Unknown
Source)
 at java.util.Scanner.nextInt(Unknown
Source)
 at java.util.Scanner.nextInt(Unknown
Source)
 at Test.main(Test.java:8)
```

Handling exceptions: summary

OPTION #1: CATCHING

```
public void doSomething() {
      try {
            // code that might
            // throw IOException
      } catch (IOException ex) {
            ex.printStackTrace();
            // clean up
      }
}
```

OPTION #2: PASSING ON

```
public void doSomething()
      throws IOException
{
      // code that might
      // throw IOException
}
```

Throwing an Exception

Use the **`throw`** keyword:

```
throw new Exception (“Invalid input”);
```

You can throw an Exception at any point in your code

String parameter indicates the message (available through `ex.getMessage()`)

If you throw a checked Exception, you also need to add it to the header of your method with the **`throws`** keyword

```
public String processInput (String input) throws Exception { … }
```

Advantages of using Exceptions

1. Separating out error-handling code

   Instead of a series of if/then/else statements

   Just “assume” that things will work and deal with errors elsewhere

2. *Propagating* errors up the call stack

   i.e., sending errors along until they reach a method that is prepared to handle them

3. Grouping error types

   Exception is a class, and can be subclassed

   => different types of Exceptions can be conceptually grouped together

   (I/O exceptions, for example)