CITS5501 Software Testing and Quality Assurance
Exceptions and where to throw them

Unit coordinator: Arran Stewart

Overview

This lecture contains a very brief overview of when and how we should throw exceptions.

Some of it you should already be familiar with from previous units that covered object-oriented programming.

Off the "happy path"

When we write or use a method or function, there's often a "typical" or "normal" or most likely case that can occur.

For instance, consider code for opening a file and reading a text file in Java:

```
import java.nio.file.Files;
import java.nio.charset.StandardCharsets;

// ...

List<String> lines = Files.readAllLines("myfile.txt",
                                        StandardCharsets.UTF_8);
```

The "typical" case is that the file `"myfile.txt"` exists, and we have permission to read it, so a list of lines will be returned.

Off the “happy path”

But it’s very easy to predict that in some cases, the file *won’t* be there, or we won’t have permission to read it. Java uses the `IOException` class to inform the caller about these situations:

```
public static List<String> readAllLines(Path path, Charset cs)
    throws IOException
```

But they’re such common and predictable situations that arguably, they’re not very “exceptional”. So maybe we shouldn’t be using exceptions to inform the caller about them at all.

Languages without exceptions

In fact some languages (like Rust), exceptions don't even exist, and the return type of a function like `readAllLines` would be more like:

```
EITHER a list of lines OR an IOError
```

But Java doesn't support an "either" type like this, so we must make do with exceptions.

Off the "happy path"

So this is one case where we use exceptions: to inform the caller of a method or function about a perfectly predictable circumstance. If such circumstances arise, they don't indicate a logic error or other problem with the calling code.

(Python does this a lot – every time you iterate over a list, under the hood, Python excepts the list to throw a `StopIteration` exception when the end of the list is reached.)

Those exceptions and when they will be thrown **should** be properly documented, so the caller knows what kind of situations can occur and handle them all.

Easily predictable situations that are off the "happy path"

- Throw an exception (if that's the idiom for the language you're working in) to inform the caller
- Document what exceptions are thrown and when. These form

Unfixable situations

Sometimes, situations will arise where a system cannot continue normal operation, and also can't reasonably do anything to fix the situation.

In Java, these situations are indicated using the `Error` class:

> *An* `Error` *is a subclass of* `Throwable` *that indicates serious problems that a reasonable application should not try to catch. Most such errors are abnormal conditions. The* `ThreadDeath` *error, though a "normal" condition, is also a subclass of* `Error` *because most applications should not try to catch it.*

Unfixable situations

Examples of `Errors` in Java include `VirtualMachineError` (the Java VM has run out of resources, or its state has become corrupt).

In general, we should **not** try and catch these (since there's nothing we can do to the fix them), and often it's not possible to predict when they might arise. (We certainly shouldn't *throw* them ourselves; that's the JVM's job.)

Unfixable situations

- It's not our job to throw these (but do check out the Java assert statement, which lets us document invariants and indirectly could throw them)
- We shouldn't try to catch them
- Since we generally can't predict them, there's no point documenting them in our methods (how could we know they'd be thrown?)
- All we can reasonably do is let the program abort execution
- Such errors don't form part of the expected behaviour of the system

Unfixable situations – a footnote

As a footnote – sometimes we may be working on platforms with constrained resources,[1] or on a hard real-time system.

In those situations, it might be reasonable to do something with these sorts of errors (or we might be able to better predict when they could arise).

But such platforms are out of the scope of this unit; if you're ever working on one of them, you should consult the documentation and industry best practices for how to deal with these sorts of errors.

[1] For instance, JavaCard, a version of the JVM designed to be run on smart cards.

Other situations

So we've seen we could have

- perfectly predictable but atypical situations
  - we should throw exceptions to inform the caller
  - we should document the exceptions and write tests to ensure they're thrown when they should be
- unfixable situations
  - it's not our job to throw or catch these

But there are a few situations that do fall outside of these two categories.

Preconditions

What if we write a `Date` class with the following constructor:

```
/** Creates a Date which represents a date in the proleptic
 * Gregorian calendar on or after 1st January, 1 CE.
 *
 * @param year The year, which must be greater than 0
 * @param month The month, which must be in the range 1—12
 *    inclusive
 * @param day The day of the month, a number between 1—31
 *    inclusive;
 *    it must be less than or equal to the number of days in
 *    <pre>month</pre>.
 */
public Date(int year, int month, int day)
```

Preconditions

```
/** Creates a Date which represents a date in the proleptic
 * Gregorian calendar on or after 1st January, 1 CE.
 *
 * @param year The year, which must be greater than 0
 * @param month The month, which must be in the range 1—12
 *    inclusive
 * @param day The day of the month, a number between 1—31
 *    inclusive;
 *    it must be less than or equal to the number of days in
 *    <pre>month</pre>.
 */
public Date(int year, int month, int day)
```

There are preconditions for this constructor, because not all possible values of `year`, `month` and `day` make sense.

Suppose a caller breaches the preconditions; should we throw an exception?

Preconditions

The answer is “It depends”.

We know that we don’t *have* to throw an exception – if the caller breaches the preconditions, it’s their own fault and they deserve whatever happens.

But should we check, and throw one anyway?

(Note that if we do throw an exception, we don’t have to document it, either. And also note that this isn’t an “atypical but expected case”; if the caller passes in invalid values, there’s some sort of logic error in their code.)

Preconditions

We *could* add a Java `assert` statement to our constructor code:

```
public Date(int year, int month, int day) {
  assert isValidCombination(year, month, day);

// ...
```

We define a helper method to check whether the parameters supplied form a valid combination for a `Date`, and we `assert` that the combination is indeed valid.

`assert` statements

```
public Date(int year, int month, int day) {
  assert isValidCombination(year, month, day);

// ...
```

In Python, C++ and many other languages, this would be the *best* solution. We `assert` things that *should* be true; if they aren't, there's some sort of logic error in the (callers' or our) code.

An `AssertionError` gets thrown, and execution of our program aborts (which is usually the best thing to do, if we've entered an erroneous state).

`assert` in Java

Unfortunately, by default in Java, `assert` statements have no effect – it's necessary to pass the `-ea` or `-enableassertions` flags to the JVM for them to do anything.

If we're writing a library, we can't know whether `-ea` was passed.

So to get around this limitation of the language, we might decide to throw an exception of our own.

This is acceptable. Assertions would be *better*, but since they don't work reliably, throwing an exception halts execution of the program and prevents any inconsistent state from spreading further.

Question: should we test our code to ensure the exception *is* thrown when invalid arguments are supplied?

Other things that we could do

Here are a few other things we might do some (or multiple of) when encountering an exceptional situation we can't handle:

- do nothing
- log a warning or error message
- abort execution by calling `System.exit()`, allowing cleanup of resources
- send the JVM a `SIGKILL` signal,[2] which cannot be handled or ignored, and results in immediate termination of all threads, with no cleanup actions executed

Which of them do you think would be best, and under what circumstances?

[2] See Roel van de Paar, "How Linux Signals Work: SIGINT, SIGTERM, and SIGKILL".