CSSE2002/7023

Programming in the large

Week 3.2: Exceptions

This hour:

- Exception example
- Throwing exceptions
- Exceptions and inheritance
- Pros and cons of exceptions
- Collections
- java.util.Stack
- Primitive wrappers

Exceptions

`exc1.java`

- Don’t just squash exceptions
- Once an exception has been thrown, it will unwind the stack until caught. Return does not happen.
- `finally` happens whether or not an exception was caught
- Trigger an exception with `throw`.
- A `try` can have multiple `catch` blocks

If they aren’t caught …

Let’s have `exc1` raise a `java.io.IOException`.

If they aren't caught ...

Let's have `exc1` raise a `java.io.IOException`.

If Java knows that some types of exceptions *could be thrown*, it insists you do something about them. You must either:

1. `catch` it
2. Declare that the method could throw

If they aren't caught ...

Let's have `exc1` raise a `java.io.IOException`.

If Java knows that some types of exceptions *could be thrown*, it insists you do something about them. You must either:

1. `catch` it
2. Declare that the method could throw

Use `throws`

Inheritance and Exceptions

Exceptions are objects (and hence described by classes). Eg:

```
try {


} catch (IOException e)  {
    // FileNotFoundException,
    // UnknownHostException
    // EOFException, ...
}
```

Inheritance and Exceptions

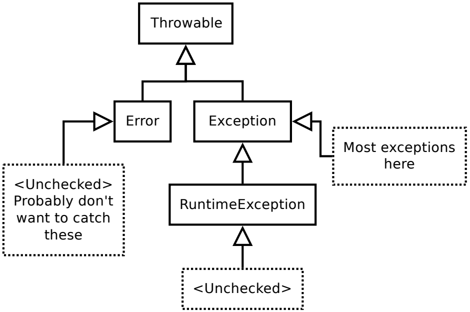
```
try {

} catch (FileNotFoundException e) {
    // This could execute
} catch (IOException e)  {
    // this could execute
} catch (EOFException e) {
    // this will not execute
    // already dealt with by
    // superclass above
}
```

Be sure to put the most general class last.

Exception heirachy

In `java.lang`:



You don’t need to declare methods throw things which are subclasses of `RuntimeException` or `Error`.
You could `catch Throwable`. Don’t! Errors are generally very bad.
What about `catch Exception` — need a good reason.

Pros and Cons

- The code that detects the problem may not know what it should do about it (move IO to borders of the program).
- Exception propagation means decisions can be made elsewhere (without needing to code a return path all the way back).
- Can carry a lot of information
- Can’t just be ignored (unless squashed)
- Java likes them
- Did something go wrong (waves vaguely) somewhere in there.
- Not as good if the problem should be checked immediately.
- Less convienient where fine control is needed
- Better for “exceptional” circumstances
- If it can be checked for ahead of time, is it better to do that instead?

CSSE2002/7023

Programming in the large

Week 3.2: Collections

Why not arrays?

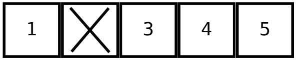
Arrays aren't great for everything:

- Fixed size at creation time — do you know how much space you are going to need?
- Don't automatically close gaps

Why not arrays?

Arrays aren’t great for everything:

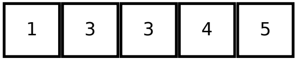
- Fixed size at creation time — do you know how much space you are going to need?
- Don’t automatically close gaps



Why not arrays?

Arrays aren’t great for everything:

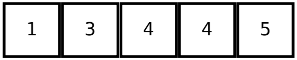
- Fixed size at creation time — do you know how much space you are going to need?
- Don’t automatically close gaps



Why not arrays?

Arrays aren't great for everything:

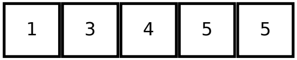
- Fixed size at creation time — do you know how much space you are going to need?
- Don't automatically close gaps



Why not arrays?

Arrays aren’t great for everything:

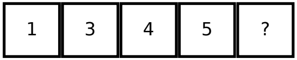
- Fixed size at creation time — do you know how much space you are going to need?
- Don’t automatically close gaps



Why not arrays?

Arrays aren’t great for everything:

- Fixed size at creation time — do you know how much space you are going to need?
- Don’t automatically close gaps



Other options

We will talk about some other datastructures. We are looking at their API (what they can do) not (unless otherwise indicated) their performance (how fast they do it)[1].

- stack
- vector
- list
- set
- map

All of these live in `java.util.*`.

[1]See an algorithms course for that.

Stack / L.I.F.O

`empty()`	is this stack empty?
`peek()`	Return the object at the top of the stack.
`pop()`	Remove the object at the top of the stack and return
`push(obj)`	Put `obj` on the top of the stack.

Stack — generics

Originally Java collections stored `Object`s.
We often want to be more restrictive (so that we don't get `Car`s in our stack of `Cat`s).
To declare a stack of `String`s:

```
Stack<String> s=new Stack<String>();
```

Stack

`Stackdemo.java`

What is it good for?

- Reversing things?
- Putting something aside to come back to later while you deal with something else now.
- “Depth-first” algorithms

Primitives in Collections

What about `Stack<int>`? No. Collections will only store Objects.
Java has a class[2] for each primitive type:
`Boolean, Byte, Character, Double,`
`Float, Integer, Long, Short`

So you can have `Stack<Integer>`.

`IntStack.java`

Autoboxing — create objects in the background. Primitives are still not objects.

[2] In `java.lang`