INFO1103: Introduction to Programming

School of Information Technologies, University of Sydney

Week 6: Exceptions and File I/O

We will cover: Need for Exceptions, `try`/`catch`/`finally`, passing the exception, binary and text files, writing to and reading from a file, streams

You should read: §§9.1, 9.3, §§10.1 – 10.3

Lecture 11: Exceptions

Your friend when thigngs go baddly
rongQ#((Q#%_)U((__))..*

Run-time Errors

Whenever a Java program deals with the external environment, many things can go wrong

- unavailable resource
- not allowed to change
- wrong format when reading
- index out of bounds
- etc.

How will the program cope?

- have many different return values, indicating error conditions; then have the calling program test for each?

Exceptions are a mechanism to deal with such cases in a much cleaner way

What is an exception?

An **exception** *is an object that signals the occurrence of an* **unusual** *event during the execution of a program.*

It's a kind of error message that is passed around around a program when something's gone unexpectedly wrong.

Passing an exception is called "throwing" it.

Under normal circumstances, exceptions are very rarely thrown: mostly, things work as expected.

Exceptions in the real world

In your everyday life you may come across exceptions in a similar way to the way in which programs meet them.

Imagine you're a postal worker…

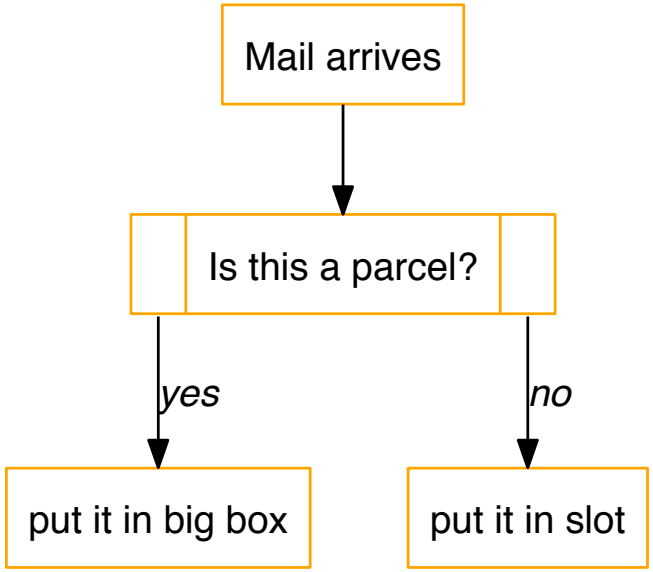
Exceptions in the real world

In your everyday life you may come across exceptions in a similar way to the way in which programs meet them.

Imagine you're a postal worker. . . . . and your name is Alfred.

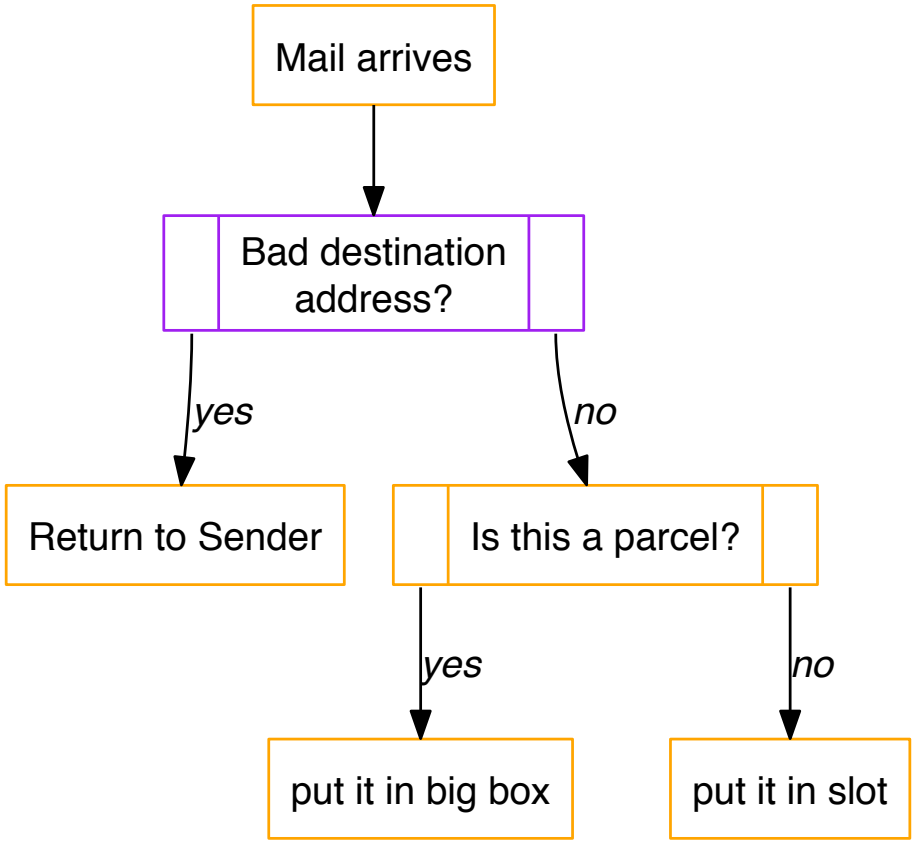
Handling the mail

Mostly, Alfred's job was simple.



Interesting Times

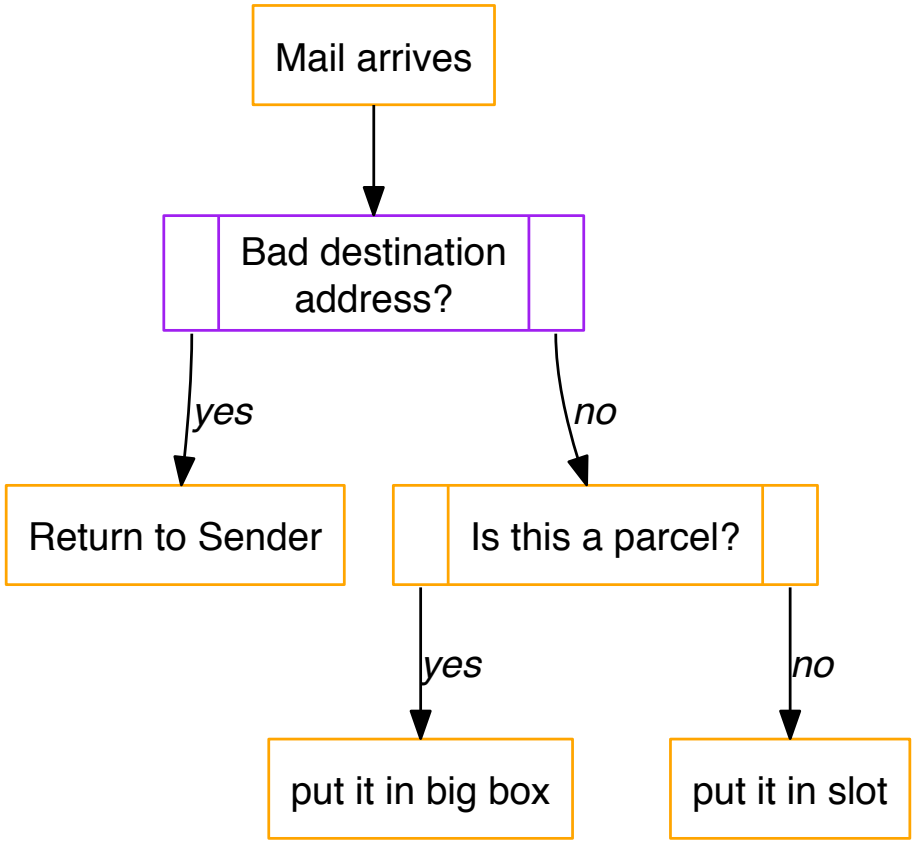
But sometimes, things were a bit more interesting.



Then, he had to think quite hard to know what to do, but he could handle it.

Interesting Times

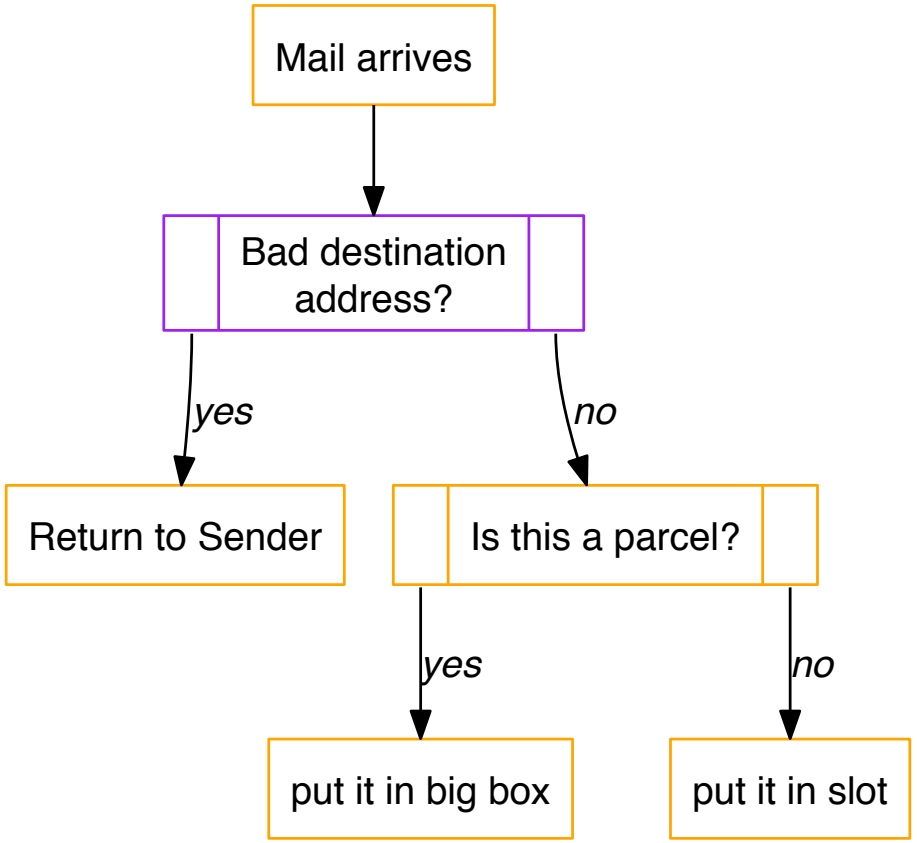
But sometimes, things were a bit more interesting.



Then, he had to think quite hard to know what to do, but he could handle it.
These circumstances were still “normal”. It wasn’t unexpected, and it was definitely something he could handle.

Interesting Times

But sometimes, things were a bit more interesting.



Then, he had to think quite hard to know what to do, but he could handle it.

These circumstances were still "normal". It wasn't unexpected, and it was definitely something he could handle.

Exceptions are for when you *can't* handle it.

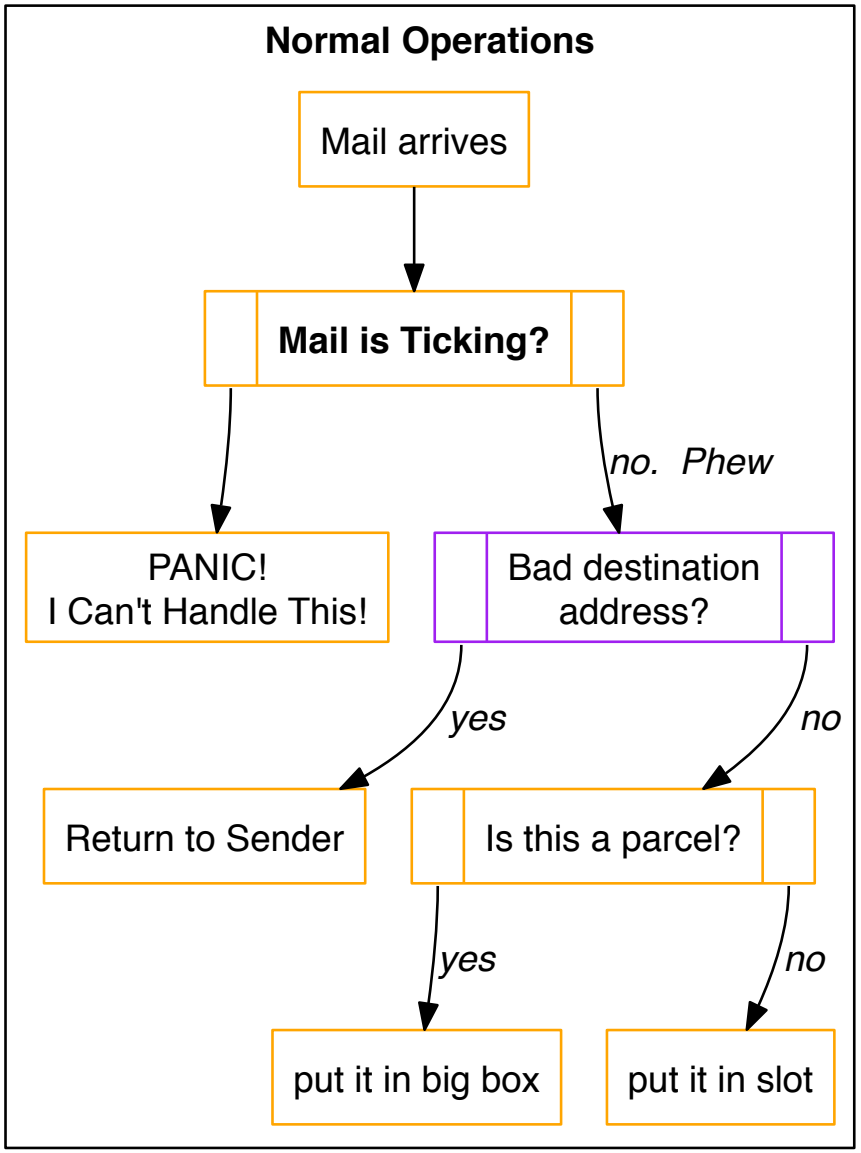
Until one day...

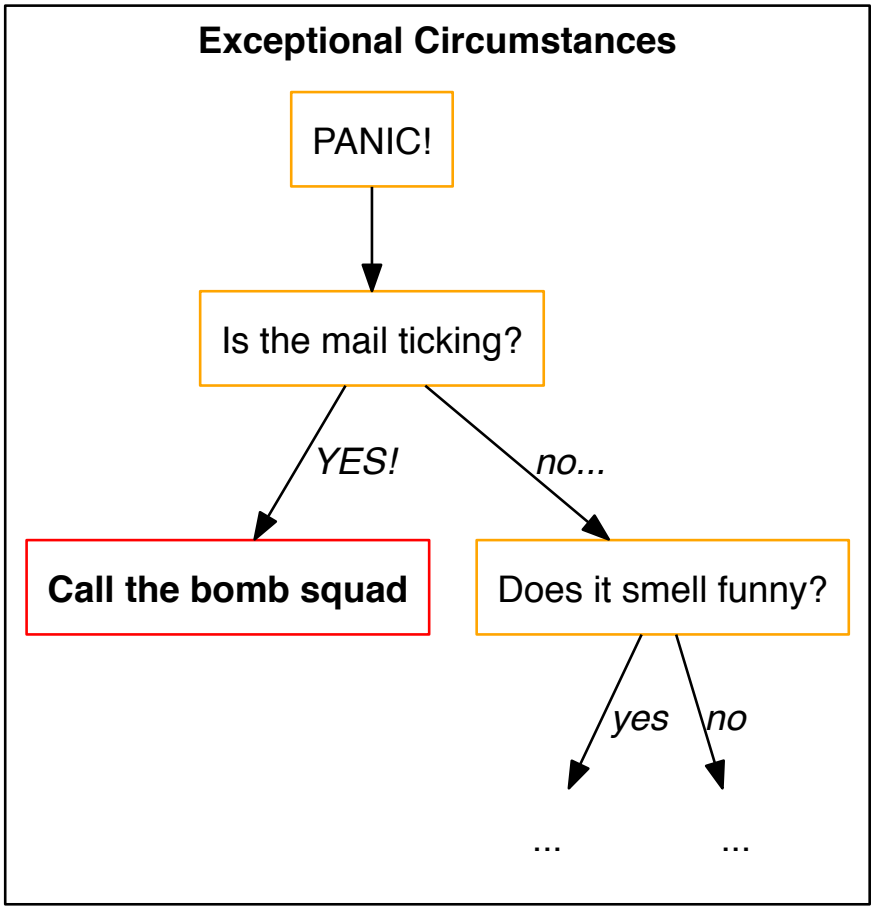
The mail.

Until one day...

The mail.

Is ticking.


Normal Operations
Mail arrives
Mail is Ticking?
no. Phew
PANIC!
I Can't Handle This!
Bad destination address?
yes
no
Return to Sender
Is this a parcel?
yes
no
put it in big box
put it in slot


Exceptional Circumstances
PANIC!
Is the mail ticking?
YES!
no...
Call the bomb squad
Does it smell funny?
yes
no
...
...

Using exceptions

Exception mechanism:

When an (exceptional) error is encountered

1. *Construct* an exception Object
   (which contains information about the error);
2. *Throw* (also called "raise") the exception;
   This stops normal flow of control.
3. The exception may be *caught* somewhere and dealt with; otherwise the execution stops.

Exceptions syntax

`try` a piece of code that contains a potential rare error

if the rare error occurs a new exception is generated. `throw`

The exception is caught using `catch` and handled

Following the `try`, error or not, more code can be executed with a block `finally`.

Exception *throw*

Code that can't handle a strange error should throw an exception:

```
1    // never expect this situation
2    if ( numberOfGPU < 1 ) {
3       throw new Exception("Cannot find any GPU. Cannot play.");
4    }
```

`Exception` is a class in Java. You need to make an Object with the `new` keyword.

`Exception` is very general, it does not indicate what the specific problem is.

Which Exception do I use?

RuntimeException is a kind of exception that can be thrown during normal running of the Java Virtual Machine (JVM);

These are kinds of (= "subtypes of") RuntimeException

IndexOutOfBoundsException when attempting to read "outside the boundary" of an array;

ArithmeticException an operation that is undefined, divide by zero, modulus zero (1 % 0)

NullPointerException when trying to use an object that is not initialised and is therefore `null`, before it is passed to a method;

FileNotFoundException self explanatory

ConcurrentModificationException when trying to modify a collection of objects (e.g. an `ArrayList`) *while* moving through it;

There are 73 Exceptions, 48 RuntimeExceptions...No.

Run-time exceptions

Why specialise the Exceptions?

Throwing a particular kind of exception means that the kind of error that caused it can be used to solve the problem

When generating an Exception, be specific as possible e.g. `new IndexOutOfBoundsException()`. Recall the Exception is an object that can be passed around. The error handler is not always *near* the error itself.

When to throw an exception

You might throw an exception when:

- an error or a situation occurs that makes the normal flow of processing unsuitable;
- 'someone else' should handle the error.

```
// ax^2 + bx + c = 0
// prints roots
public static [type] quadraticRoots
    (double a, double b, double c)
    throws ArithmeticException
{




    ...
}
```

Catching Exceptions

When code catches an exception,

- the error can be dealt with, and execution may continue, or
- it can clean up the damage and exit.

```
String input = "10a";
try {
   int requests = Integer.parseInt( input );
   int i = 0;
   while (i < requests) {
     dostuff(i);
     i++;
   }
 } catch (NumberFormatException e) {
    System.err.println("invalid number");
 }
```

Exceptions — structure

Exception-handling code

- always begins with a `try` block, which should surround all the code you think should normally work – but which could throw an exception;
- must be followed by one or more `catch` blocks *in increasing order of generality*: that is, from the most specific to the most general;
- may be followed by a `finally` clause, which contains code that will always be executed if it's there;
- should not be used for control flow!

Some exception code

Separate the "normal case" code (in the `try` block) from the code for weird situations (the "`catch`" block(s)).

You may have several *different* catch clauses, depending on kind of exception thrown in the `try` block.

The optional `finally` block is always executed at the end of the catch region:

- after the `try` block completes successfully,
- or after a catch block is executed.

```
try {
// do something
// which could throw
// Exceptions
}
catch(EOFException e) {
// do something to deal
// with the situation
}
catch(FileNotFoundException e) {
// do something else
}
catch (Exception e) {
// do something general
}
finally {
// code that should happen
// after normal/weird case
}
```

Exception catching – example

```
1    try {
2        double[] roots = quadraticRoots(a, b, c);
3        ...
4    }
5    catch(ArithmeticException e) {
6        System.out.println("Oops: " + e.toString())
7    }
```

You can refer to the exception, which we've called `e`, in the catch block.

Passing it on

If an exception occurs in a method

- constructed and thrown explicitly
- or raised by some other method that is called,

if it cannot be handled within the method then that method will *terminate* and throw the same exception to the caller of the method.

The exception will be passed up until someone can *handle* it.

Passing exceptions to calling method

Methods should *declare* the exceptions they can throw:

```
1    public void read() throws IOException
2          { read data from a file }
```

This is *enforced* for a checked exception

the compiler will check that this kind of exception will be *caught* somewhere by a method (calling a method) calling this one.

Checked and Unchecked Exceptions

Java has *two* main kinds of exceptions: *checked* and *unchecked*.

Checked exceptions are checked at compile time to see they are handled in some way: caught or thrown further.

Unchecked exceptions occur at run-time and are often (but not always) the result of *programmer error*

Don't just catch "`Exception`", particularly not to shut the compiler up!

Compilers help check exceptions

```java
import java.io.IOException;
public class Exceptional {
    public static void foo() throws IOException {
        // don't do anything in fact
    }
    public static void bar() {
        foo();
    }
    public static void main(String  [] args) {
        bar();
    }
}
```

```
~> javac Exceptional.java
Exceptional.java:7: unreported exception java.io.IOException; must be caug
      foo();
         ^
1 error
```

Checked Exceptions

Checked exceptions

- force tighter coding;
- are useful especially for situations out of programmer's influence (e.g., when programming in groups!)

Unchecked exceptions

- are often produced by programmer errors in ordinary code, e.g.,
  - NullPointerException
  - IndexOutOfBoundsException
  - ClassCastException

The “throws” header

Here’s how you declare what exceptions a method can throw:

```
    public void getData(File infile) throws IOException
    {
        read data from a file
    }
    public void dealWithData(File f)
    {
        try {
            getData(f);
            // some other stuff
        }
        catch (IOException e) {
            System.out.println("Oops: faulty datafile?");
        }
    }
```

The compiler checks that all methods calling `getData` either handle an `IOException`, or are themselves called by something that does.

Common `Exceptional` errors

- Putting `try/catch` blocks with too small scope: put them around the entire piece of code you expect to work!
- Returning too general an Exception type: use the appropriate exception if you can.
- Not conveying information with the exception thrown: construct it with a detailed message.

Principles

Do not *throw* an exception

- to avoid thinking about flow control in typical situations. e.g. don't deal with the end of the input this way!
- to "simplify" your code!

Do not *catch* an exception

- if it is not the right place to handle it — maybe let the calling routine handle it;
- to shut the compiler up!

Use exceptions in exceptional cases.

Summary

- `try ... catch ... finally ...`
- Java has *checked* and *unchecked* exceptions.
- Checked exceptions which are not caught internally must be mentioned in method header throws clause.
- Exceptions are not suitable for expected situations flow-control!
- Catch and handle the exception if you can do so sensibly, else pass it on.

Throwing an exception — example

```
public class WeightCheck
{
   public static void printWeight(double aWeight)
      throws IllegalArgumentException
   {
      if (aWeight < 0.0) { // expecting non-negative
         throw new IllegalArgumentException("negative weight");
         /* this location can never be reached */
      }
      /* code here for the normal case */
      System.out.println("weight is: " + aWeight);
   }
   public static void main(String[] args) {
      try {
         printWeight(-2);
      } catch (IllegalArgumentException iae) {
         System.out.println("Arguments invalid");
         System.out.println("Exception message: " + iae.toString());
      }
   }
}
```