Prifysgol Abertawe
Swansea University

CS-230 Software Engineering

L18: Using and Designing Exceptions

Dr. Liam O'Reilly

Semester 1 – 2020

What are Exceptions?

- Exceptions are used to model errors or bad situations.

- When a bad situation is detected an exception can be created an thrown.
- This stops the current execution and allows an exception handler to handle (catch) the error.

- Most library methods will throw exceptions.
- You can and should throw exceptions too!

Using Exceptions

Two Aspects to Exceptions

- There are two aspects to dealing with bad situations at run time:
  - Easier Part: Code must be written to detect errors:
    - Use if statements to test variables and decide if there is a bad situation.
    - If there is then we can throw an exception.
    - Most library methods will already do this.
  - Harder Part: Handling Errors:
    - You need to 'catch' each possible exception and resolve the bad situation.
    - You should only catch exception if "you" are in a position where you ca actually deal with them.

Example: Detecting a Bad Situation

Imagine we are writing code for an Cash Machine (i.e., a ATM)

- We want to allow withdrawing of money only when there is enough money in the account.

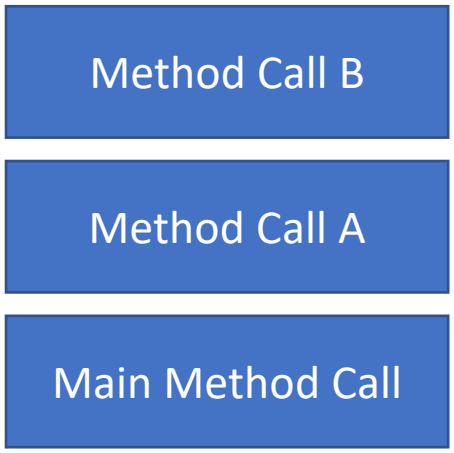
```
public void withdraw(Account a, int poundsToWithdraw) {
    if (a.getBalance() < poundsToWithdraw) {
         // Not enough money in account
         throw new IllegalArgumentException("Insufficent funds");
    }

    // Code to actually withdraw money goes here.
}
```

- We detect the bad situation and throw an exception.
- If we throw an exception the method is aborted.
- There are many “types” of exceptions – choose appropriate ones.

What Happens to Thrown Exceptions

Imagine a program with a main method that calls method A. Method A then calls Method B.

- This will form at runtime a call stack.



Lets assume that code inside Method B throws an exception.

- If that code is in a try block, then the catch block deals with it.
- If not, then control flows down the stack. If the code in Method A that called B is in a try block then the exception is handled.
- If not, then control flows down the stack. If the code in Main Method that called A is in a try block then the exception is handled.
- If not, then program crashes.

Example: Handling a Bad Situation

```
// Code to ask user how much money

try {
    withdraw(yourAccount, requestedAmount);
} catch (IllegalArgumentException e) {
    // Nice message to user explaining what went wrong
}
```

- We catch the exception and print a nice error message.

- We could even enclose the whole try-catch block in a loop to allow the user to re-enter the requested amount.

To Catch Or Not To Catch

- We want to call a method that might throw an exception.
- Now we have a major choice:
  - Use a try-catch block to handle the error.
  - Leave a possible exception flow down the call stack.
- Only catch an exception if “you” are in a position where you can handle the bad situation properly!
  - Really properly!
- It is better not catch the exception if you cannot handle it.
  - Leave it to the calling code to handle the exception.

Example: To Catch Or Not To Catch

```
public void withdraw(Account a, int poundsToWithdraw) {
   if (a.getBalance() < poundsToWithdraw) {
       // Not enough money in account
       throw new IllegalArgumentException("Insufficent funds");
   }

   // Code to actually withdraw money goes here.
}
```

This method cannot possibly handle the bad situation. It is not its fault that there is not enough money in the account. It is the callers fault for attempting to withdraw too much money in the first place.

Thus, it should throw the exception!

Actually, the message in this exception could be better!

It could include the balance of the account and the amount requested.

E.g., “Insufficient funds. Account balance 54.76 but 65.00 requested.”

Example: To Catch Or Not To Catch (2)

The calling code is a fault (in this case). It asked the user for how much money they wanted to withdraw.

It is in a position where it can handle the bad situation properly.

It can report an error message to the user and allow them to re-enter the requested amount.

```
// Code to ask user how much money

try {
    withdraw(yourAccount, requestedAmount);
} catch (IllegalArgumentException e) {
    // Nice message to user explaining what went wrong
}
```

Throw Early – Catch Late

- Throw Early:
  - When a method detects a problem that it cannot solve, it is better to throw an exception, rather than try to come up with an imperfect fix.

- Catch Late:
  - Only catch errors if you (the code) are in a position where you can actually remedy the bad situation properly!
  - If you catch an error then the error should be dealt with and execution continues as normal.
  - If you are not in such a position, the best action is simply to have the exception propagate to its caller, allowing it to be caught by a competent handler.

- Better to crash the program than continue on with even more errors due to bad situations not actually being resolved correctly.

Squelching Exceptions Hurts!

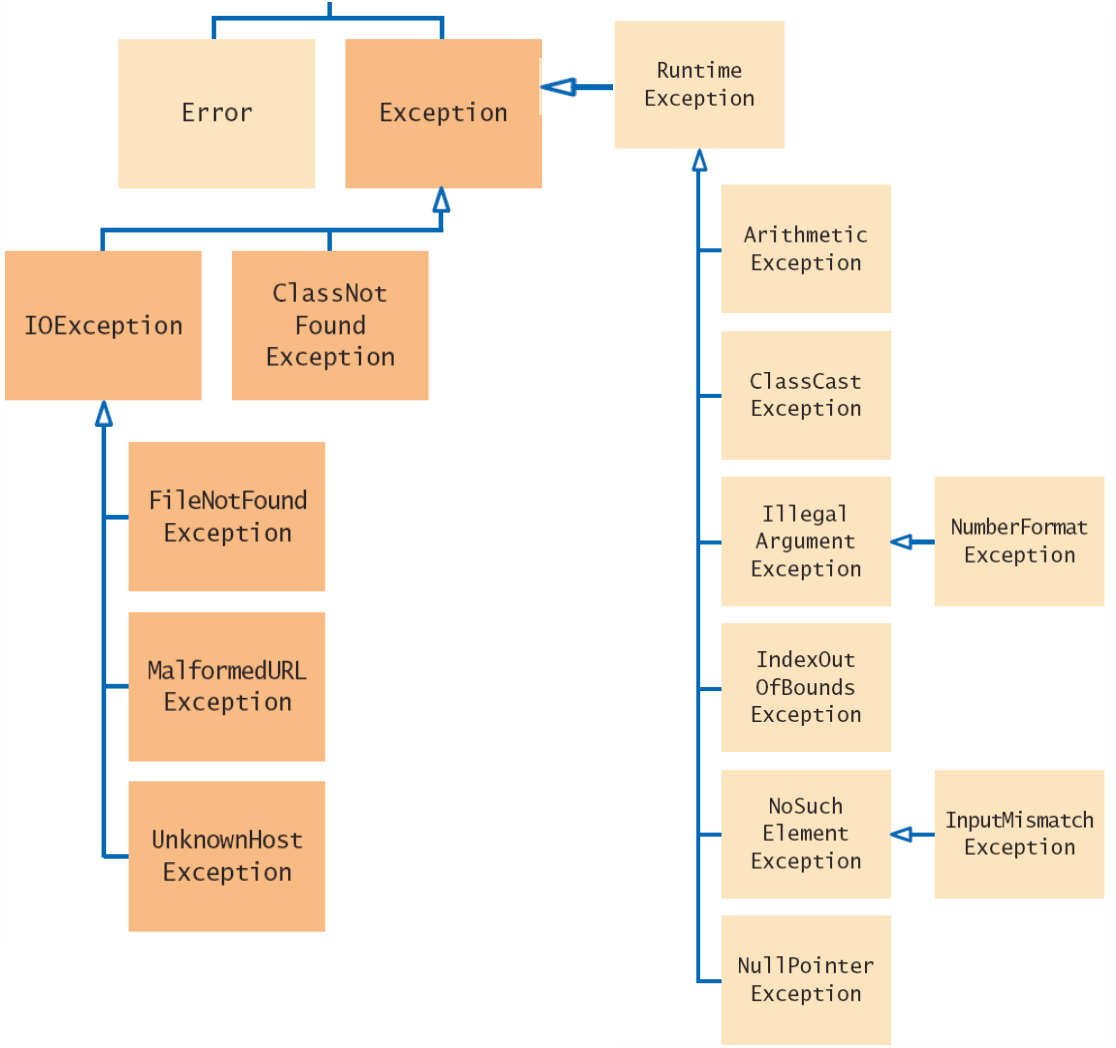
- Squelch Exceptions:
  - Basically means using an empty catch block.

```
try {
    // Code that might throw an exception
} catch (Exception e) {
    // No code here
}
```

  - Why might you do this?
    - When you call a method that throws a checked exception and you haven't specified a handler, the compiler complains.
    - It is tempting to write a 'do-nothing' catch block to 'squelch' the compiler and come back to the code later. Very bad Idea!
    - Exceptions were designed to transmit problem reports to a competent handler.
    - Using an incompetent handler simply hides an error condition that could be serious.

Types of Exceptions

- Pick appropriate exceptions to throw.
- Java has a whole hierarchy of them.
- Always provide a nice message string in the constructor.



IllegalStateException: Used when class is not in an appropriate state to carry out a method.

E.g., Mario. Mario object cannot jump if he is not on the ground.

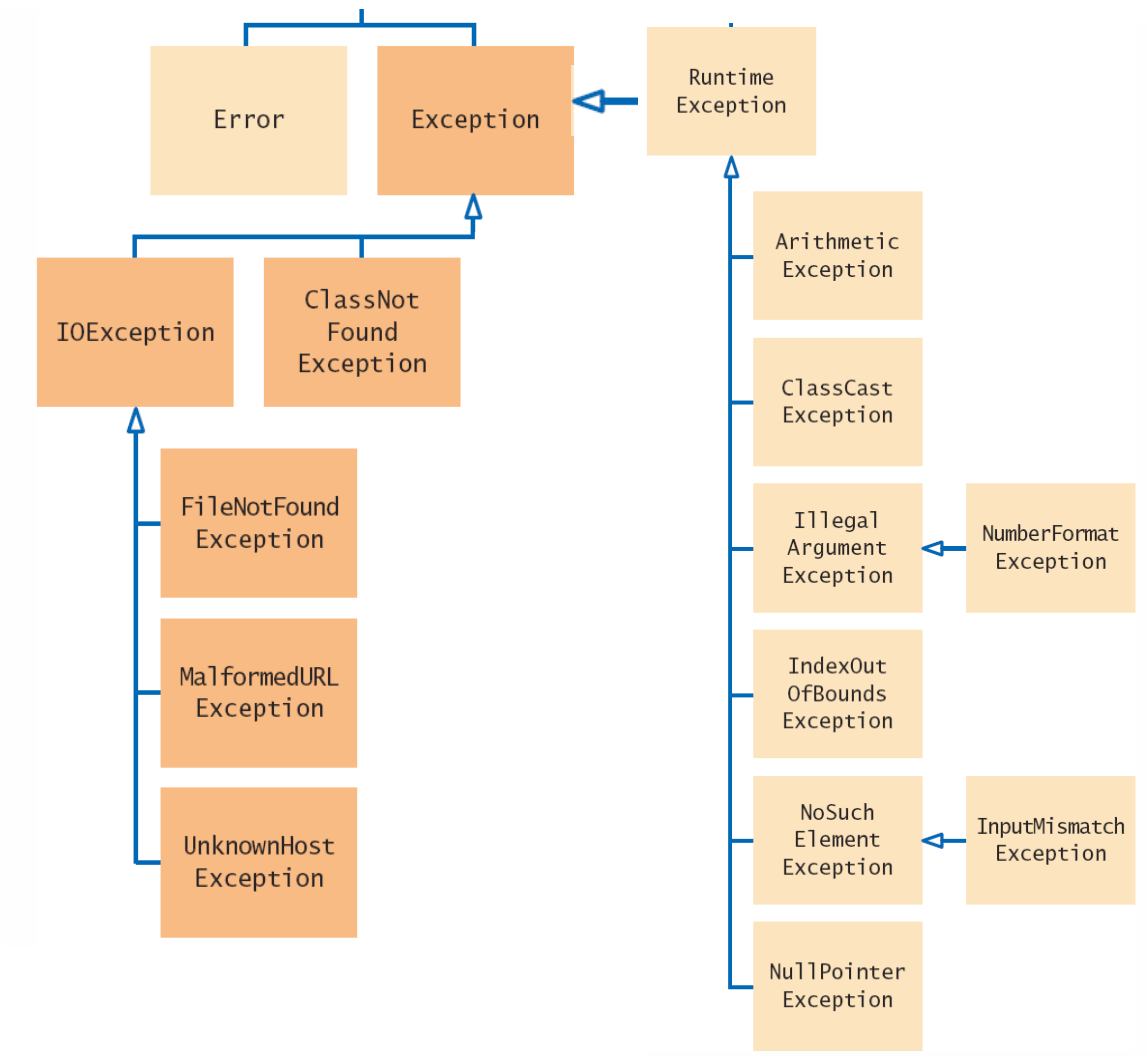
IllegalArgumentException: Used when data being passed to method is not appropriate.

Example: Method to calculate area of circle, but negative radius passed in.

Designing Custom Exceptions

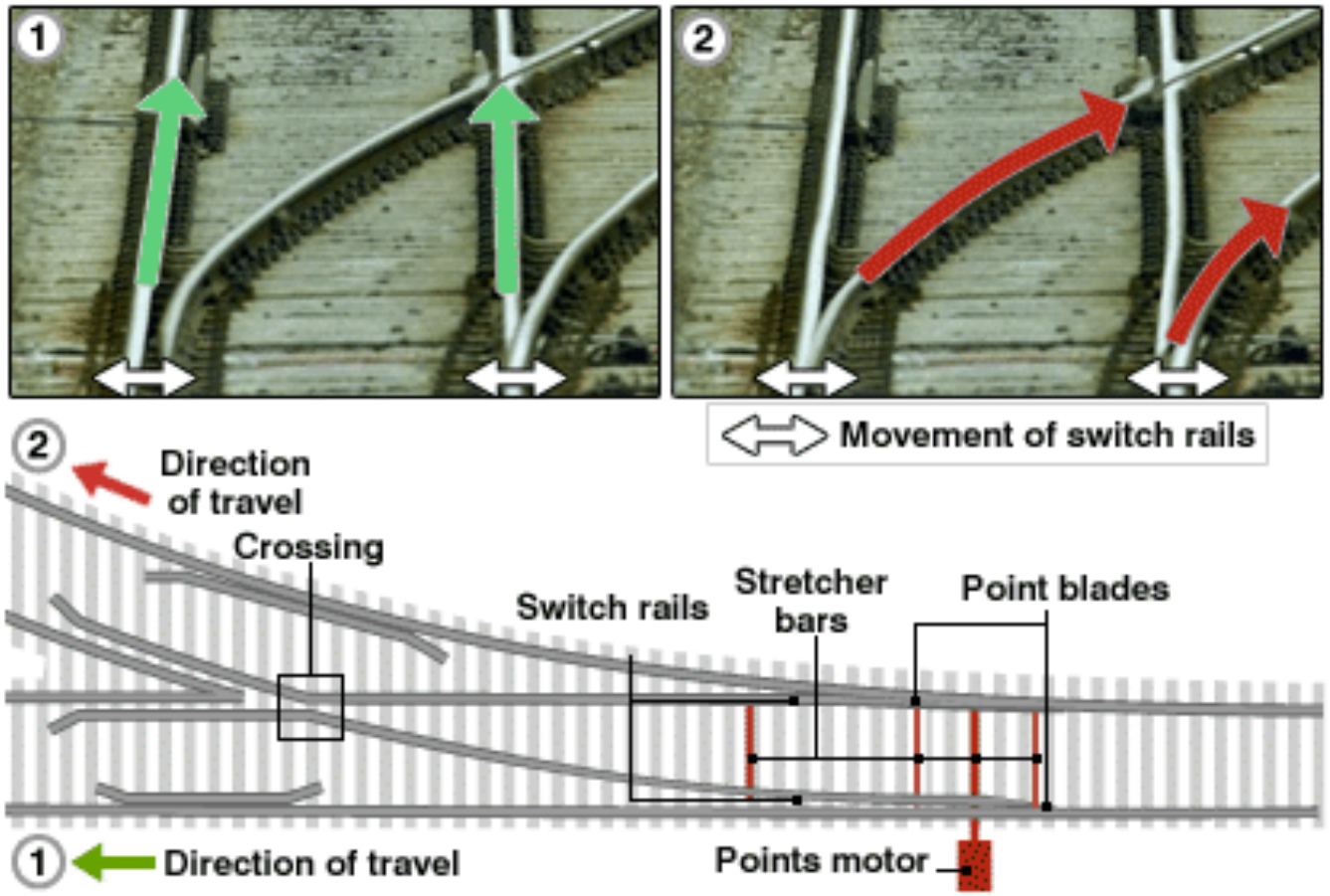
Creating Your Own Exception SubClasses

- There is no reason why you cannot create your own exception classes.
- Easy to do.
- Just find the best current match and create a subclass.
- You can then throw your new custom exception (and catch them too).



Example: Railway

- Railway systems use points to allow trains to change tracks.



Example: Interlockings

- Manual Interlockings in signal boxes used to control them.
- These days computer systems control the points.
- In both cases, there are many situations where a point cannot be changed (i.e., set):
  - A train is occupying the track.
  - The point is part of a track circuit that is locked (and allocated to a train).
  - Flank protection from other near by trains.

Example: Using Custom Exceptions

```
public class Track {
    proetcted String trackID;
    protected boolean isOccupied;

    ...
}
```

```
public class point extends Track {
    ...
    public void setPoint(boolean normalState) {
        if (isOccupid) {
            throw new PointOccupiedException("Point " + trackID +
                "  + is occupied and can not be set".)
        }
        // Change point
    }
}
```

- We can throw more meaningful exceptions (like above).
- Makes the code more self-documenting.

Example: Creating Custom Exceptions

```
public class PointOccupiedException extends IllegalStateException {

    public PointOccupiedException(String message) {
        super(message)
    }

}
```

Creating our own exceptions is easy:

- We don't need to do much at all.

- Just pick an appropriate exception and create a subclass.
- All the exception constructors take a message string as a parameter.
- We just pass this along to the super constructor.

- You could of course add more attributes and behaviour to your exceptions if you like.
  - But that would be a more advanced setup.

Summary

- Exceptions model error situations.
- Throwing them is easy.
  - You should do as much checking as you can and throw exceptions as soon as you detect bad situations.
- Catching them is hard:
  - You have to think about the whole architecture of your application.
  - Only catch them if “you” (i.e., a method) are in a position to actually handle and resolve the bad situation.
  - If you are not then leave the exception propagate down the call stack.
- Making your own exceptions is easy.
  - Can add clarity to your program.