Exceptions

What exceptional things might our programs run in to?

Exceptions do occur

Whenever we deal with programs, we deal with computers and users.

Whenever we deal with computers, we know things don't always go as we'd like.

Whenever we deal with users, we know that users don't always behave as we'd hoped for.

When we write code, we also deal with programmers. We know that programmers don't always *do the right thing*™

How do we deal with these cases?

Different types of exceptions

Some things might happen which we couldn’t really predict. A network failure, a crashed hardware etc.

Some things are more of a contingency (eventualities) nature. You don’t have enough money to make an order, you are not allowed to do something right now.

From these two types of extreme, unwanted situations arises a need to programmatically deal with these situations.

Contingencies - things that might occur

If we know that on rare occasions, an action will not work for logical reasons which are part of the rules for our application, it would be good if we could deal with these cases in a structured way.

Let’s say that we have a rule for a system for a webshop which says that a customer cannot place two orders within five seconds (because this indicates fraud or unnatural behavior). Then the method for processing orders could revoke an order in an orderly fashion, by “throwing an exception”.

Code calling the processOrder() method should of course be required to deal with this rare eventuality.

A checked exception

Programmatically, in Java, we would then declare that the method “throws” a so called “checked exception”.

Exceptions thrown are just objects which are of some class extending one of the Exception classes in the Java API.

Exceptions which do not inherit from Error or RuntimeException (but rather from Exception or some of its descendants) are called checked exceptions, because a method which throws such an Exception must declare that it does so (or handle it internally). And code calling a method which declares that it might throw a checked exception must write code to deal with it (or rethrow it).

Exceptions

An exception is what we call an exceptional event which occurs when our program is running, and which interrupts the normal flow of the program's instructions.

This is important. Wherever an exception occurs, the program flow is interrupted and the control moves up the call chain until the first code which can "catch" the exception and regain control.

a() -> b() -> c() -> d()

If an exception occurs in d(), and only a() has code catching it, control will move up to a(). Catching the exception *could* be handled in any of them.

```
public class CallChain {
  public static void main(String[] args) {
    a(); // Start of call-chain
  }
  static void a() {
    System.out.println("Inside a(), Trying to call b()");
    try {
      b();  // a()
    } catch (Exception e) { // Handler!
      System.err.println("a(): Caught exception: " + e.getMessage());
    }
  }
  static void b() throws Exception {
    System.out.println("Inside b(), Calling c()");
    c();    // a()->b()
  }
  static void c() throws Exception {
    System.out.println("Inside c(), Calling d()");
    d();    // a()->b()->c()
  }
  static void d() throws Exception {
    System.out.println("Inside d()"); // a()->b()->c()->d() we are now at the bottom
    throw new Exception("Somthing went bad in d()");
  }
}
```

```
$ javac CallChain.java && java CallChain
Inside a(), Trying to call b()
Inside b(), Calling c()
Inside c(), Calling d()
Inside d()
a(): Caught exception: Somthing went bad in d()
```

The process order thingy

The method processOrder() from our web shop example could then declare that it throws an exception of some type. The type could be TooQuickOrderException (which might extend Exception).

The code calling the processOrder() method, must write special code to be prepared for the event of a TooQuickOrderException being thrown, or there would be a compilation error.

This way, Java has a mechanism for forcing programmers to be aware of contingencies (eventualities) and also handling them.

Runtime exceptions

Another main type of exceptions in Java are the runtime exceptions (descendents of RuntimeException).

A runtime exception often indicate a programming error, and often has no resolution. Trying to follow a null reference to an object and invoking a method is something a programmer could do, but there is no way to recover from that situation. The program would behave erroneously.

Another typical runtime exception is when a method tries to divide a whole number with zero. This is not possible and would result in a runtime exception. There is no obvious way to recover from that situation.

```
String name = null;
System.out.println( name.toUpperCase() ); // will crash!
// NullPointerException will be thrown, since name is null

int a = 10;
int b = 0;
System.out.println(  a / b  ); // will crash
// ArithmeticException will be thrown, because division by 0

String name = "Ragnar Rök";
Passport pass = (Passport)name; // will crash
// ClassCastException, because we can’t convert a String to
// a Passport!
```

More runtime exceptions

If a programmer tries to cast an object to a class which is not a class in the correct hierarchy (e.g. trying to cast a Passport to a Motorcycle), a ClassCast exception would be thrown. This is a programmer error and there is really no way to recover from this exception.

Programmer errors are bugs and should be detected and fixed.

Can all checked exceptions be recovered from?

An interesting fact is that all checked exceptions, that is, exceptions that the compiler enforces you to handle, can't really be recovered from.

Let's say that your application should fetch the current weather from an internet service and display a forecast to the user.

If the internet service is down (or you don't have a network connection for some reason), what would the recovery really be (other than an error message)?

If your application tries to read a file, but the filesystem is broken, the same applies... what can your program do about it?

We'll talk more about checked and runtime soon

In a coming lecture, we'll discuss the two main types checked exception and runtime exception more in detail.

For now, it is enough that you think about the concepts as two main types:

Checked: need to be handled programmatically (hopefully you can do something about it)

Runtime: doesn't require handling but will make your application fail which sometimes is the right thing (if the exceptions comes from bad coding)

Some exceptions are severe failures

Let’s take an address book application in a mobile phone as an example.

If the addresses are saved in a database on the memory card, and the phone cannot find the memory card, is there any point of keeping the application running?

This indicates that sometimes we have the need for some global exception handler code which can deal with really bad things and exit the program nicely (e.g. with an error message explaining what happened).

“Sorry, the memory card is missing or broken. The address book will close.”