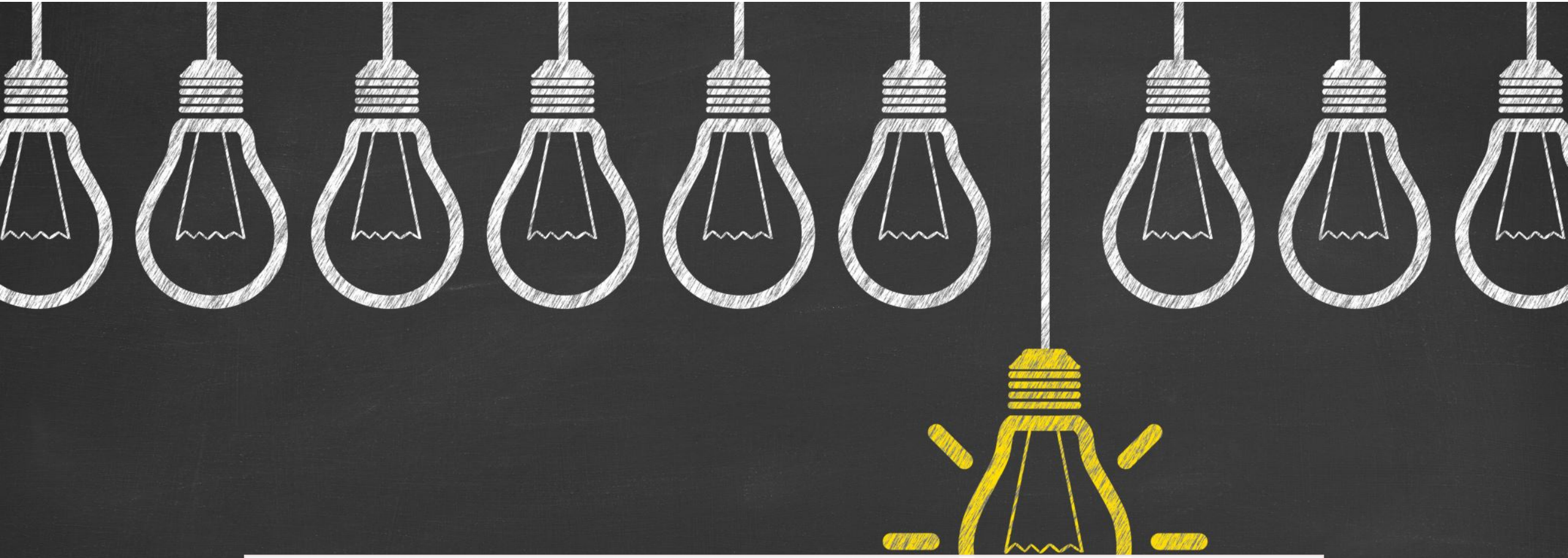


Exceptions II

CS 209 A ~ Week 2 Presentation 1

Exceptions Part 2

As we have seen last time, you can either catch exceptions, or pass them back to the caller. If it's a checked exception and you don't catch it, you must declare that the method can throw the exception.

But… Something else can happen

Compile-time
Link-time
Run-time
Logic

- What if you have logic errors in your program?
- Java cannot see these types of errors

Program terminates with an incorrect result

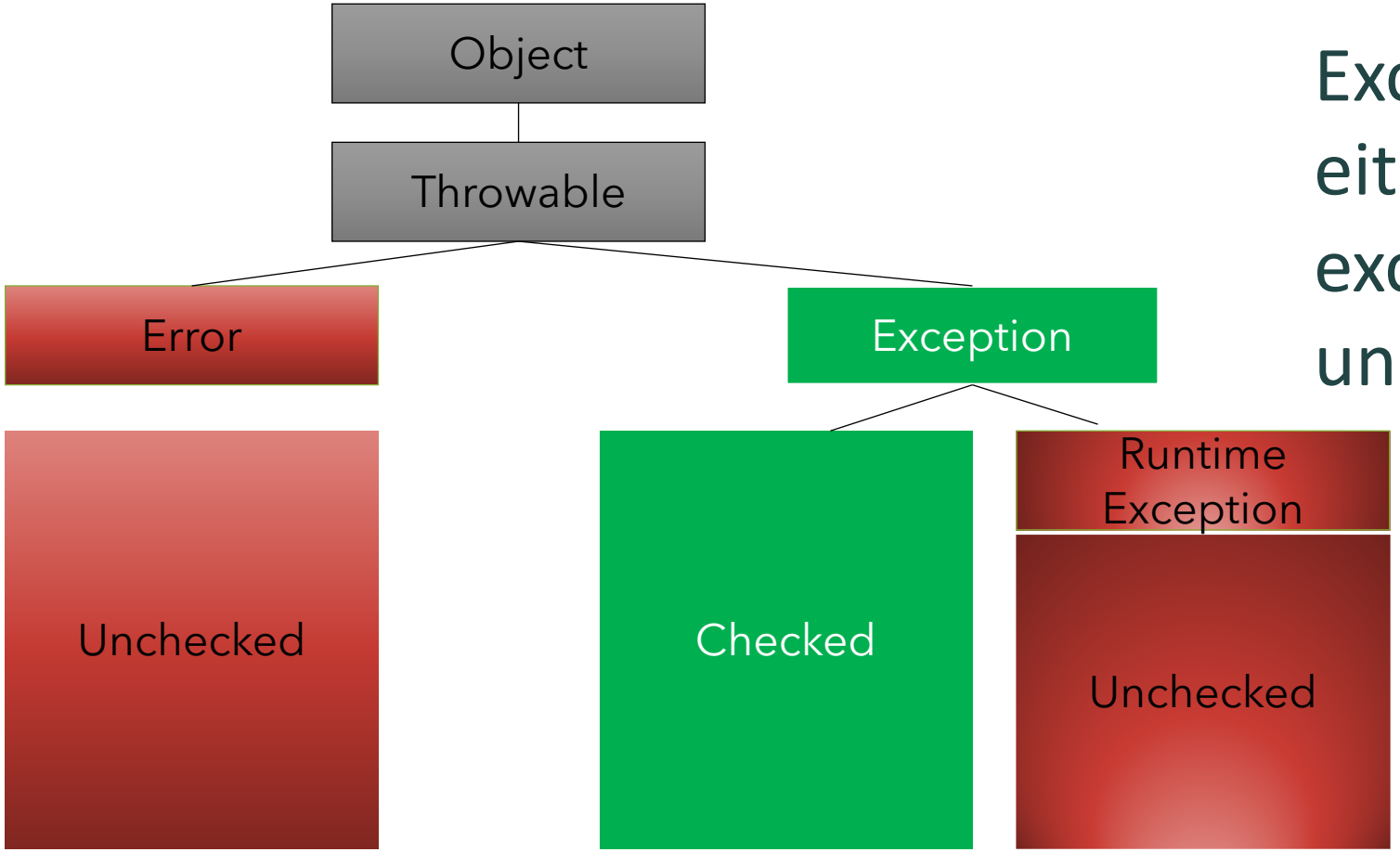
Program crashes because of divide by zero error in a calculation
Or… Null object references
Or… Access an array with an out of bounds index

Logic Errors

- These types of errors are not really expected happen
- They are called "**Unchecked Exceptions**"
  - Errors
  - Runtime exceptions

No requirement for catching or declaring

Checked and Unchecked Exceptions



Exceptions can be either checked exceptions or unchecked exceptions

Unchecked Exceptions

- A method is not forced by compiler to declare the unchecked exceptions thrown by its implementation. They will come into life once buggy code is executed.
- Examples: ArithmeticException, ArrayStoreException, ClassCastExcepti on

Why is this important?

This is important because you often want to create your own exceptions in an application when the application logic / business rules are violated (for example in a bank application a negative account balance may generate an exception).

➢ *You can create your own exceptions*

```
…
} catch (MyException e) {
    …                                   … handling
}
```

Creating New Exception Types

- You don't create your own exceptions from scratch but extend a Java Throwable so you have to decide what class you will extend.

```
class MyException extends Exception {
 …
}
```

Checked

- **Choices**: should you extend Exception? Then it will be a checked exception and javac will ensure it is used according to the rules…

Creating New Exception Types

Unchecked Exceptions are subclasses of RuntimeException

```
class MyException extends RunException {
 …
}
```

Not Checked

- **Choices**: Or will you extend RuntimeException and then javac will not force users of your exception to follow the requirements of declaring it in the method name and using throw or try/catch

Is an event an exceptional event??

- Depends how often the event/case happens…

- Unlikely? “once a week?”
- Shouldn’t happen? “once every 5 years?”

Care is necessary: for instance in big transaction system one in a million may be every day

Always check?

In theory, checked is better as javac will ensure that all the users of your method work properly. However in practice it may be different; if you are modifying an existing class already used in dozens of programs, you don't want to add an exception that will require all these programs to be modified (and tested) to take it into account (you can also assume if everything has run without this exception so far, it can go unchecked). Or you may want to provide a "wrapper" method that catches the exception.

Is it reasonable to recover?

- One convention for deciding between checked an unchecked exceptions is for **checked** exceptions to be used where the case is *predictable but also unpreventable* **and** *reasonable to recover from*

- **Unchecked** exceptions should be used for everything else

Predictable but also unpreventable

Method caller did validate the input parameters however some condition outside their control has caused the procedure to fail.

<u>Example:</u> attempting to load file that has been deleted it between the time you check if it exists and the time the read operation commences. Here by declaring a checked exception, you are telling the caller to anticipate this exceptional case

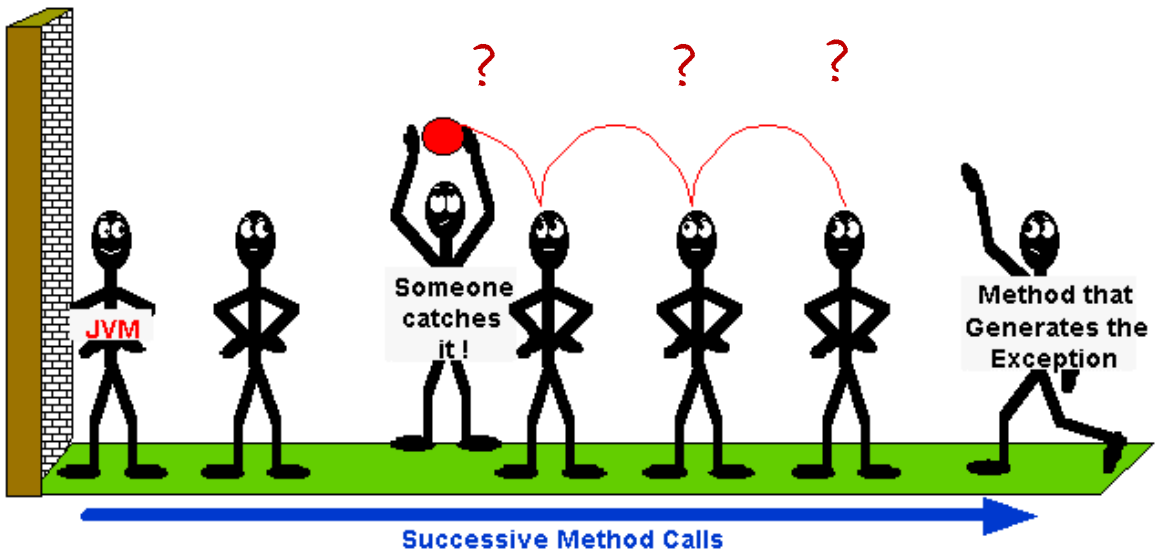
Reasonable to recover from

It is pointless to force callers to anticipate exceptions that they cannot recover from.

If a user attempts to read from a non-existing file, the caller can prompt them for a new filename. On the other hand, if the method fails due to a logic error arising from a programming bug (invalid method arguments or buggy method implementation) there is nothing the application can do to fix the problem in mid-execution.

One thing could do is to log the exception and report to support team (more later).

Who catches it?

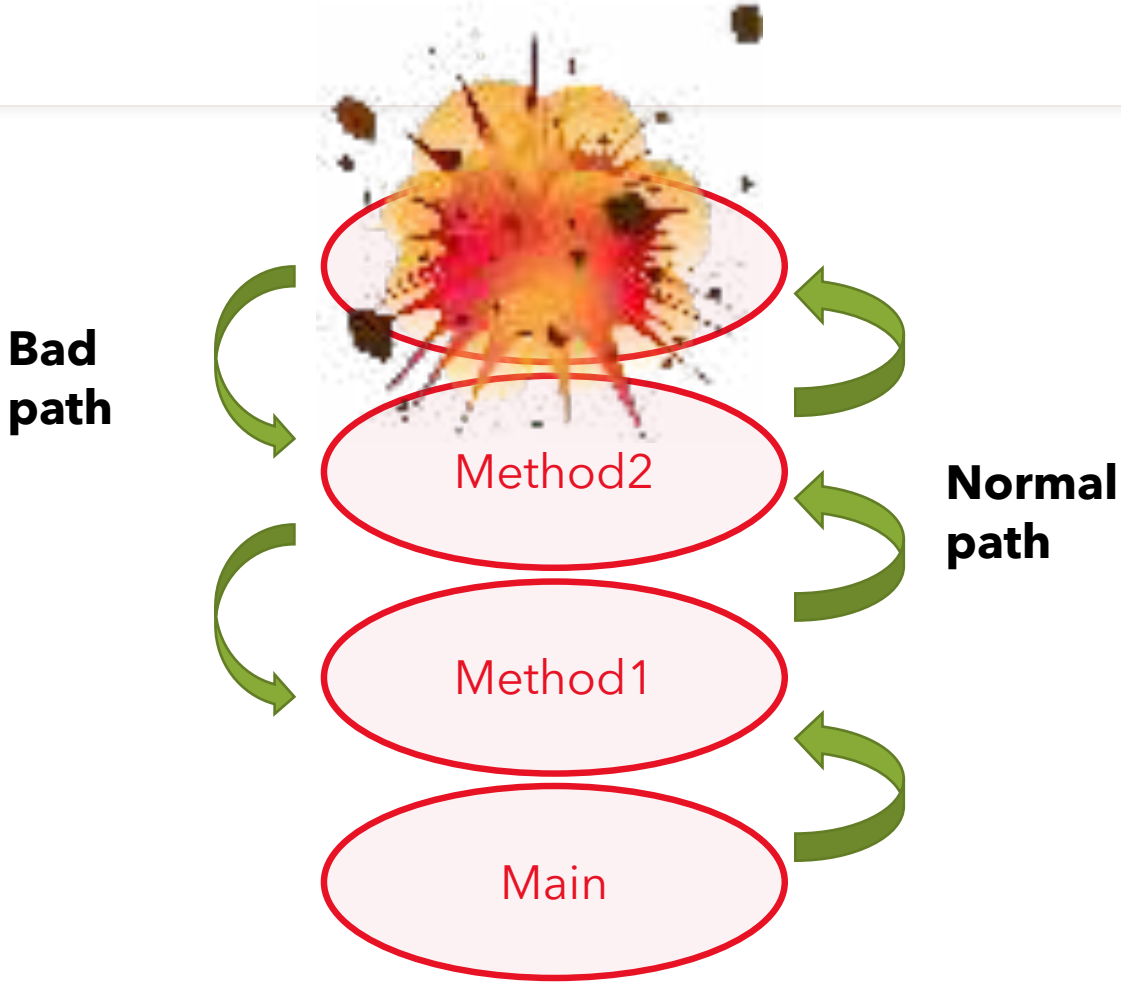


Choose the right abstraction level

Exception wrapper

For both checked and unchecked exceptions, **use the right abstraction level**. For example, a code repository with two different implementations (such as a database and a filesystem storage) should avoid exposing implementation-specific details by throwing SQLException or IOExc eption. Instead, it should wrap the exception in an abstraction that spans all implementations (e.g. RepositoryException).

Exception Processing



Whether your exception is checked or unchecked, you have to take into account what happens when it's thrown.

Exception Processing

```
try {
…
        WalkDownStairs();
} catch (ExceptionType fallover) {
…
        GrabHandRail();
}
```

- Everything is fine case
- And if you catch the exception you need to thing about what to do when things turn bad as they often do.

Exception Processing

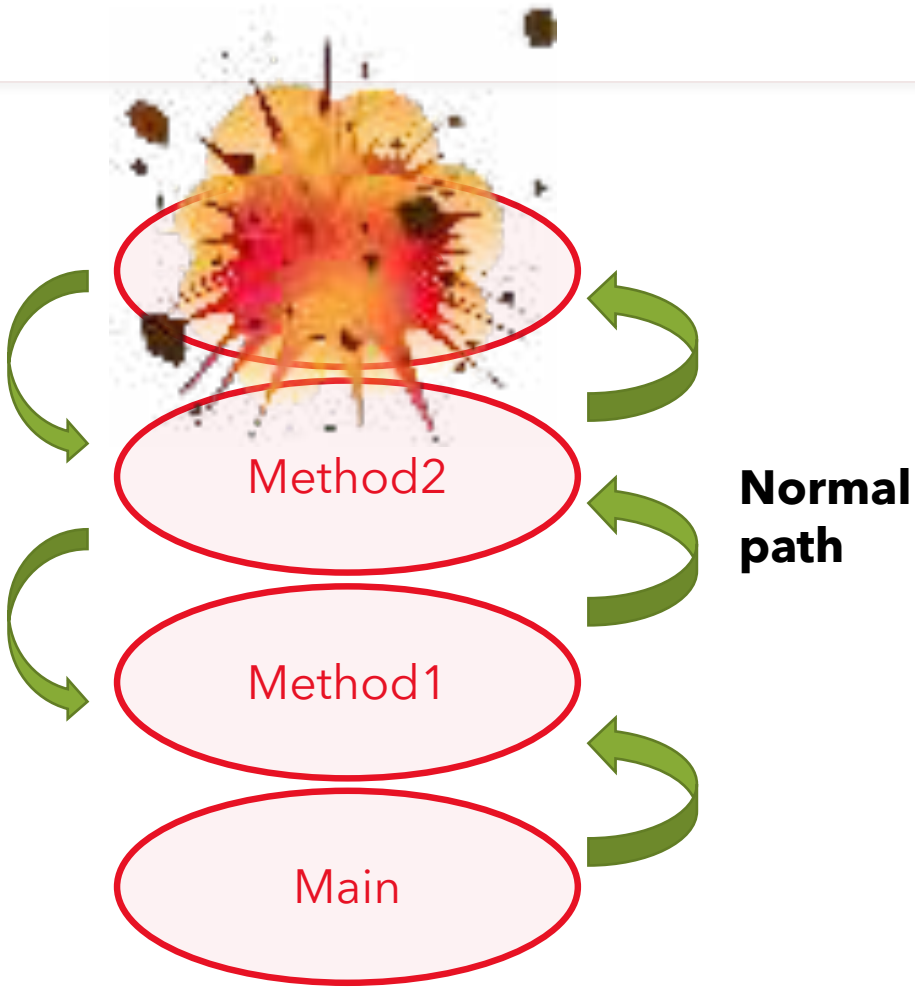
```
try {
        …
} catch (ExpectedException e1) {
        …
} catch (ExpectedException e2) {
        …
} catch (…)
        …
}
```

It can become complex, especially when deciding about how the application should behave next.

Unknown unknowns

- Unexpected and not known or checked
- Also called “Unknown unknowns”

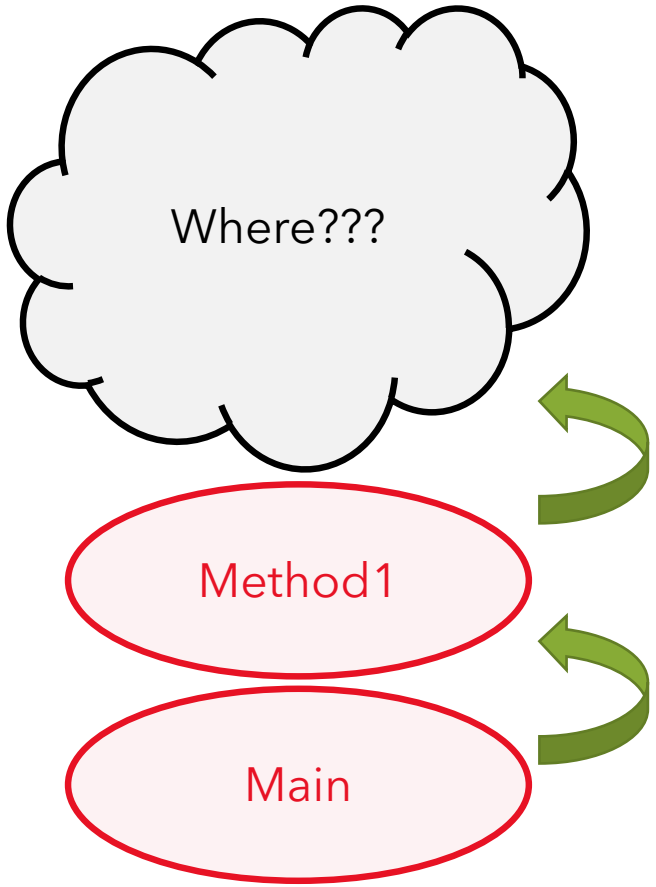
Exception Processing



One possibility is to just ignore exceptions and let them "bubble up"

Here we let method1 deal with the problem that happened in method3

The problem




```
} catch (Exception e) {
        System.err.println("Error: %s", e.getMessage());
...
}
```

- The problem is errors that happen too often have to be fixed
- Here method2 might call many methods apart from method 3 and we have to find where the problem is

Throwable

Throwable()

Throwable(String message)

Throwable(String message,
Throwable cause)

Throwable(Throwable cause)

String	getMessage()
Throwable	getCause()
Void	printStackTrace()

The throwable class has constructors that can help to make it easier to track down errors

NOTE: these constructors are in Throwable and not in the Error or Exception classes

Handling unknown unknowns

Option 1:

Just ignore them – one possibility is when something really unexpected happens just ignore it

Handling unknown unknowns

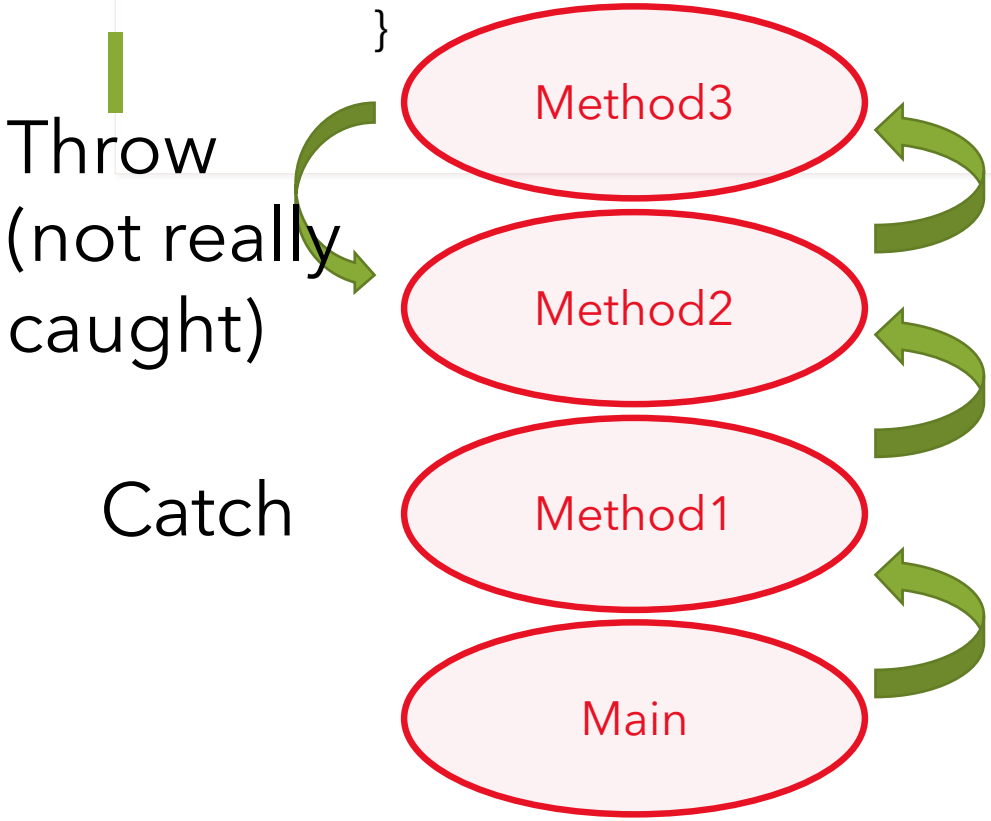
Option 2:

Pass up information about the exception and how it occurred (cause)

Processing exceptions

- A method can catch any exception and then throw it again with a message that says the exception was noticed

```
throws Exception
} catch (Exception e) {
        throw new Exception ("method2", e)
}
```

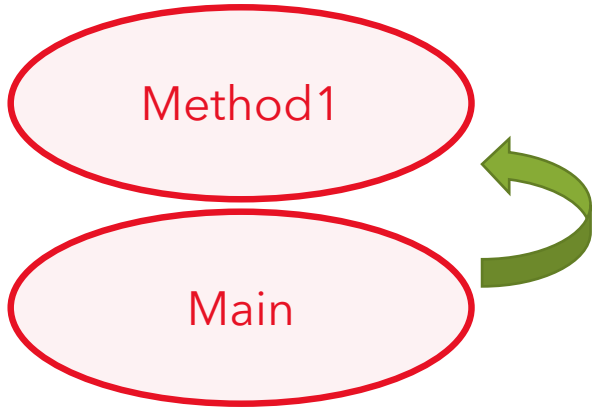


Chained Exceptions

```
} catch (Exception e) {
        System.out.println("Error: " + e.getMessage());
        Throwable t = e.getCause();
        while (t != null)
                System.out.println("Cause: " + t.getMessage());
                t = t.getCause();
        }
        …
```

- When we catch the exception we can display the full list of things that wen wrong

Catch



Exception Processing

```
try {
        …
} catch (ExpectedException e1) {
        …
} catch (ExpectedException e2) {
        …
} catch (Throwable t)
        // unknown and unexpected
        …
}
```

You could ignore exceptions by catching them and doing nothing

NEVER IGNORE

the unexpected

You should only ignore things that you know aren't important – for instance, getting past the end of a string when you are done scanning it ...

Error Messages

- Don’t **SCARE** end users with excessive technical detail in messages

- $ java Prog
  Enter an integer value: A Exception in thread "main"
  java.util.InputMismatchException
- at java.util.Scanner.throwFor(Scanner.java:864) at
  java.util.Scanner.next(Scanner.java:1485) at
  java.util.Scanner.nextInt(Scanner.java:2117) at
  java.util.Scanner.nextInt(Scanner.java:2076) at Prog.main(Prog.java:8)
- $

Error messages

- But provide enough detail for the support team

~~technical problem occurred. Please contact support.~~

.. and providing enough information either for the user to correct the problem by oneself ("Invalid number" in the previous case) or to the people who in many companies are here to help customers or colleagues. A short error code or message saying what went wrong would help them.

Finally some operations need to always be performed

```
try {
        open a network connection
        send a message
        close the connection
} catch (…) {
}
```

Finally some operations need to always be performed

```
try {
        open a network connection
        send a message
        this is the purpose of finally – it is executed in all cases
} catch (…) {
} finally {
        if connection opened, close it
}
```

Some operations have to be performed in all cases

There is a new alternate syntax in java versions greater than 7 (Java >= 7)

”try with resources”

Since Java 7, there is a new syntax known as "try with resources". You can just after try instantiate a new object. If this object has a close() method, and if it implements a special interface, close() will be called automatically at the end of the try {} catch {} block, just as if this method had been called in a "finally" block.

Declarations must implement the (auto)closable interface

```
try (Statement stmt = con.createStatement()) {
    ResultSet rs = stmt.executeQuery(query);

    while (rs.next()) {
        String coffeeName = rs.getString("COF_NAME");
        int supplierID = rs.getInt("SUP_ID");
        float price = rs.getFloat("PRICE");
        int sales = rs.getInt("SALES");
        int total = rs.getInt("TOTAL");

        System.out.println(coffeeName + ", " + supplierID + ", " +
                           price + ", " + sales + ", " + total);
    }
} catch (SQLException e) {
    JDBCTutorialUtilities.printSQLException(e);
}
```

Call.close() is executed when exiting the try-catch block

```
    try {
        Statement stmt = con.createStatement()
        ResultSet rs = stmt.executeQuery(query);


        while (rs.next()) {
            String coffeeName = rs.getString("COF_NAME");
            int supplierID = rs.getInt("SUP_ID");
            float price = rs.getFloat("PRICE");
            int sales = rs.getInt("SALES");
            int total = rs.getInt("TOTAL");


            System.out.println(coffeeName + ", " + supplierID + ", " +
                               price + ", " + sales + ", " + total);
        }
    } catch (SQLException e) {
        JDBCTutorialUtilities.printSQLException(e);
    }finally {
    try {
    stmt.close();
    } catch (SQLException e) { // do nothing

        }
    }
}
```

Assertions

Use ONLY when developing code

Finally we have to talk about "assertions". To "assert" means to state or to claim.
You claim that something is true. If it's wrong, the program crashes. When you develop, it allows you to test that you are for instance getting values in the expected range.

Assertions

Assertions allow you to check during testing that some exceptions won't occur.

claims

```
assert (val >= 0);
double result = Math.sqrt(val)
```

Assertion error

Assertions

You can also add error messages, then track why you got a value you shouldn't

Error message

```
assert (val >= 0) : val + " < 0";

double result = Math.sqrt(val);
```

Assertions

- To use assertions:

  java –ea MyProgram

- Assertions are ignored if you don't pass the –ea flag (Enable Assertions) to java.
- They are a debugging tool.