COMP6700/2140 Exceptions

Alexei B Khorev, Josh Milthorpe & Steve Blackburn

Research School of Computer Science, ANU

16 March 2017

Error Codes

It pays to expect the unexpected. A method may fail to produce the expected result, due to:

- incorrect input e.g. `sqrt(-1.0)`;
- incorrect environment e.g. missing file; or
- programmer error e.g. null pointer.

How can we write our methods so their clients will be aware that something unexpected has happened, and one must switch to “plan B”?

A simple approach (used in C and other low level languages) is to reserve special values which the method returns to indicate *exceptional* situations. These special values form part of the specification of the method, and must be documented. The client must check for special return codes, and handle them appropriately. For example, the method `InputStream.read()` returns a special value of −1 to indicate that no input is available:

```
InputStream in;
int i;
while ((i = in.read()) != -1)
   // do something with i
```

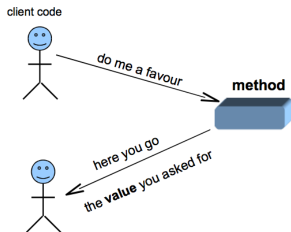
In C, a macro `EOF`, special functions like `feof` and a global variable `errno` are used to handle possible I/O errors. Regular and exceptional situations are treated as branches of the same flow of control.

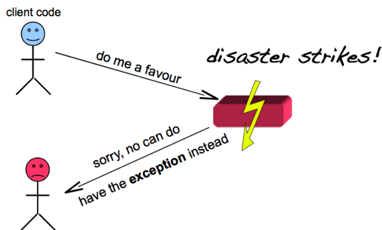
Exceptions: Control Flow for Error Handling

Mixing together code to handle both normal and exceptional situations can make a program very complicated.

Exceptions allow us to use the *type system* to separate the two.

- In normal operation, a method returns a value of its *return type*.
- How to handle the situation when a method cannot fulfil its contract?
- *Throw* a value of an *exception type*!
- Computational model is preserved. An exception becomes a part of the method signature, together with arguments and return value.





Java Exceptions

Exceptions are thrown either:

implicitly (via a program error):

```
int x = 1 / 0; // throws ArithmeticException
```

or explicitly (by executing the throw statement):

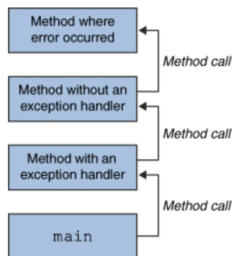
```
public double computeAgeDiscount(int age) {
  if (age < 0) {
    throw new IllegalArgumentException("Invalid age: " + age);
  // ...
}
```

Exceptions are caught with a catch block.

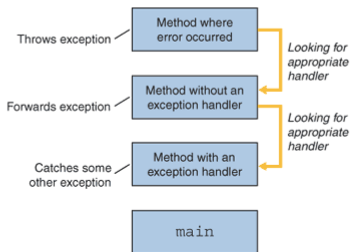
```
try {
  discount = computeAgeDiscount(age);
} catch (IllegalArgumentException e) {
  discount = 0.0;
}
```

Exception Propagation

When an exception is thrown, it is propagated from callee to caller until a matching *exception handler* is found.



Method call stack



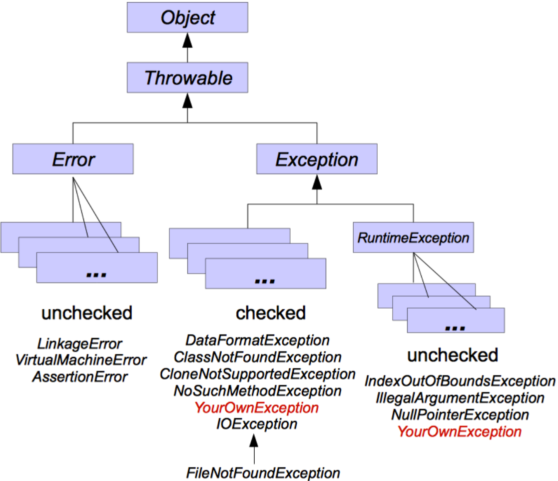
Exception goes back

Exception Types

Like everything in Java, exceptions are objects. Exceptions come in three flavours:

- *Checked exception*
  - An error from which the client could be expected to recover
  - Any code that may throw a checked exception must either *catch* it, or else *specify* the exception using a throws clause in the method declaration
  - Checked exceptions are subclasses of `Exception` which are **not** subclasses of `RuntimeException`, for example `NoSuchMethodException`, `IOException`
- *Error*
  - Irregular event that cannot be anticipated and recovered from e.g. `OutOfMemoryError`
  - Should not be declared in a `throws` clause
  - Usually not caught, as the program should terminate as soon as possible
- *RuntimeException*
  - Exception which originates within the application, but cannot be foreseen or recovered from – bugs e.g. `NumberFormatException`, `IndexOutOfBoundsException`
  - `catch` or `throws` are not required. Can be caught, but usually better to just terminate execution (and fix the bug)

Exception Inheritance Tree



Exception Handler (try-catch) Syntax

```
try {
    // do something that may generate an exception
} catch (ArithmeticException e1) { // first catch
    // this is an arithmetic exception handler
    // handle the error and/or throw an exception
} catch (Exception e2) { // may have many catch blocks
    // this an generic exception handler
    // handle the error and/or throw an exception
} finally {
    // this code is guaranteed to run
    // if you need to clean up, put the code here
}
// this is where the control flow goes after try-catch-finally
```

The order in which the caught exceptions are listed is important: if their types overlap (e.g. `FileNotFoundException extends IOException`), the narrower type must precede the wider to avoid skipping a specialised response.

An example from *Java Tutorial* for the method `writeList()` (in the class *ListOfNumbers* and *ListOfNumbers2*) demonstrates the use of the clauses in an exception handler + two examples, WithoutFinally.java and WithFinally.java.

Throwing an Exception

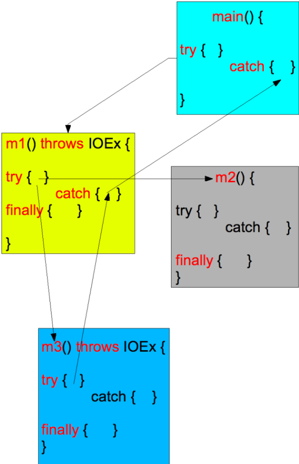
Exception objects are created when methods (or constructors) throw them:

```
public void doSomething() throws ExceptionOne, ExceptionTwo {
    // ...
    throw new ExceptionOne(); // instance of ExceptionOne class or its superclass
    // ...
    throw new ExceptionTwo(); // instance of ExceptionTwo class or its superclass
    // ...
}
```

The `throw` statement can occur either in a standard `if-else` construct (which may test for a specific state of an object), or inside the `catch` block (which will result in a chain of exceptions thrown up to the next higher level exception handler).

The exception classes listed after `throws` are the part of full method signature. If a method invokes another method which declares a `throws` clause for a checked exception, it must either catch the exception (using `try-catch` construct) or specify it in its own `throws` clause.

*Exceptiona*l chain of events: stack



*Exceptiona*l Chain of Events

This is an example of an exception chain which takes place during the execution of ExceptionTest.java. Here, exceptions throw their own exceptions (propagate sideways), some exceptions are uncaught are returned back to their caller (forwarded exceptions).

1. `main()` (inside its `try`) calls `m1()`
2. `m1()` (inside its `try`) calls `m2()`, and *then* `m3()`
3. `m2()` doesn't throw, executes its `finally()`, returns to `m1()`
4. `m3()` throws to `m1()`, executes its `finally()`, returns to `m1()`
5. `m1()` catches from `m3()`, re-throws to `main()`, executes its `finally()`, returns to `main()`
6. `main()` catches from `m1()`, terminates ("returns" to JVM)

Exceptional Issues

When an exception throwing method is overridden (in a subclass), exceptions which the superclass method has thrown can be “specialised”: i.e. the `throws` clause of an overridden method can have *fewer* types listed than the superclass method, or *more specific* types, or *both*. In other words: **the type space of exceptions becomes smaller**. It can even dispense with throwing exceptions altogether (no checked exceptions). Contrast this with “overriding” access modifiers: they can either extend method’s visibility, or leave it unchanged (`protected` or `"package"` → `public`, but not other way around).

Some programmers question (not unreasonably) the usefulness of separation checked and unchecked exceptions (see, for example, articles cited in Tim McCune’s “Exception-Handling Anti-patterns”; this paper doesn’t directly discuss this, but instead gives a good advice on when to catch an exception and when to ignore it, when and how best to define your own exception class, how exceptions are abused, and how to avoid such a practice).

Defining Exceptions

You can define your own exception class to deal with a particular situation as you see fit. By defining a checked *Exception* class, every client which invokes a method or constructor which throws your exception, must *catch-or-specify* that exception.

- It is preferable to *reuse* standard exceptions: keeps API smaller, easier for users to learn (exceptions *are* harder to learn then standard classes), less memory footprint and faster class loading (exceptions *are* expensive)
- Subclass a standard exception if you need more specific failure-capture information. and catch it first. Do not catch the generic *Exception* when you need to react to a specific *Exception* subclass. **Order `catch` clauses from more specific first to more general last.**
- Thrown checked exceptions need to be individually documented in the method's javadoc comments, `@throws` tag (required for determining which `catch` clauses to include). Documenting unchecked exceptions is also desirable since it describes the method *preconditions*.

Exceptions vs. Return Values

Only use exceptions in a truly exceptional situation, ***not*** for ordinary control flow (think first before including `try/catch` inside loops, yet see ScannerExample.java) — it worsens performance, bad for clarity.

End-of-input-stream is a regular event	**End-of-input-stream treated as exception**
`while ((token = stream.next()) != stream.END) { process(token); } stream.close();`	`try { for (;;) process(stream.next()); } catch (StreamEndException e) { stream.close(); }`

On the left, it's clear and simple. On the right, it appears to loop forever even if you know about `StreamEndException` — the loop termination condition is moved outside the loop (*SESE* violation?).

Remark: when all values read from a stream are valid (non-exceptional), it's a good idea to add an explicit `eof()` test method that should be called before reading the next token.

Guidelines for Proper Use of Exceptions

- *Unchecked run-time exceptions* are for programming errors (precondition violations etc.)
- *Checked exceptions* for recoverable conditions; they provide enough functionality (accessor methods) to gain adequate information to recover
- *Error*s are signs of resource deficiencies (memory etc), corruption of object state and other conditions which make continuation of a program execution impossible or dangerous. The rule of thumb: do not subclass *Error*.
- Checked exceptions must be handled in one or many `catch` statements — therefore, declare them sparingly to avoid putting excessive burden on their user.
- Unless an exceptional condition can be avoided through proper use of API, or API itself allows to deal with an exceptional situation, use of unchecked exceptions is more appropriate.
- The rule of thumb: never quit (`throw new AssertError()`, or `System.exit(1)`) when catching a checked exception.

Handling Multiple Exception Types

Multiple exception types can be handled by a single `catch`-statement.

The following code (taken from Evans and Verburg's book The Well-Grounded Java Developer) discriminates between exceptions caused by input error (system unable to supply the resource) or data error (missing or *bad* file):

```
public Configuration getConfig(String fileName) {
   Configuration cfg = null;
   try {
      String fileText = getFile(fileName);
      cfg = verifyConfig(parseConfig(fileText));
   } catch (FileNotFoundException|ParseException|ConfigurationException e) {
      System.err.printf("Config file '%s' is missing or malformed%n",  fileName);
   } catch (IOException iox) {
      System.err.printf("Error while processing '%s'%n", fileName);
   }
   return cfg;
}
```

Resource Management

Program resources are memory, I/O streams — opened files, sockets (network connections), pipes (program-to-program connections) and others. Memory (allocated for objects) is managed automatically by the *garbage collector*, but other resources must be cared for manually by the programmer. Correctly releasing a stream resource (file opened for reading or writing etc.) can be difficult if between opening (acquiring resource) and closing (relinquishing resource) an exception can disrupt the normal flow of control, so that the resource management remains unfinished. We can use the `finally` statement to close the resource, but... closing a stream can throw an `IOException` of its own! We must add a nested `try-catch-finally` block.

Moreover, a code with multiple external streams may include nested `try`-statements. In the example MultipleFinally.java, data is read from an `InputStream` (a *URL* connection) and written to an `OutputStream` connected to a `File`. Opening and closing each of the three can throw an exception:

- The `InputStream` can fail to open from the URL, to read from it, or to close properly;
- The `File` corresponding to the `OutputStream` can fail to open, to write to it, or to close;
- A problem can arise from a combination of factors (where most of bugs come from).

According to a JRS to Project Coin (the JDK 7 name), 2/3 of uses of `close()` in Java API (prior to JDK 7) were buggy.

Automatic Resource Management with `try(...)`

To sort out this mess, Java 7 introduced *try-with-resources (TWR)*. It allows resources to be acquired locally in the `try-catch-finally` block, so that when the block exits for any reason, the resources are relinquished. This is very similar to the way a local variable goes out of scope at the end of a block and may be garbage collected.

```
try (OutputStream out = new FileOutputStream(file);
     InputStream is = url.openStream()) {
   byte[] buf = new byte[4096];
   int len;
   while ((len = is.read(buf)) > 0) {
     out.write(buf, 0, len);
   }
}
```

The resource acquisition is performed according to statements inside the parentheses and that's it! No more care is necessary. (*Note:* chaining the stream object creation via nested constructors is not recommended in TWRs, since some streams may not be closed due to their anonymity. Instead, assign every opened stream to a variable as in the example).

If you define a class whose objects need to be "resource-managed", it needs to implement the `java.io.Closeable` interface.

Assertions: Design by Contract in Java

The state of computational environment can be characterised by an *invariant*, which is a boolean value expression that always evaluates to `true`. *Assertions* are the Java mechanism to check the invariant. It can be used at every step of computation as a test that everything goes "according to plan". If the test fails (the invariant is `false`), the `AssertionError` is thrown.

```
assert expr [:"Error message string"]; // the [...] (second expression) is optional
assert val > 0 && val < 20 : "the input value must be within the range";
```

Assertions are disabled by default. To enable them, run your application with -ea option

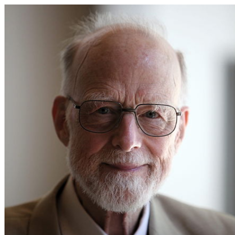
```
java -ea RootClass // enabling assertions across all classes
java -ea:package-name RootClass // enabling assertions in a package
java -ea:class-name RootClass // enabling assertions in a class
```

Assertions are turned on during development, and normally are turned off after the application is made available to customers (however, it's advisable to keep them on since they greatly simplify error analysis).

A useful "hack" — to put an assertion in a place where the execution must **never ever** get into:

```
assert false; // the code will abort if this is reached!
```

Tony Hoare and the "Billion Dollar Mistake"



Sir C. A. R. (Tony) Hoare is one of the greatest computer scientists

- In 1962 (when on a business assignment in Moscow, USSR) he discovered the *Quick Sort* sorting algorithm (and also, *Quick Select* algorithm, based on the same principle of "divide-and-concur"), arguably one of the most famous and widely used.
- In 1965, he proposed the *Hoare Logic* — an extension of the Boolean calculus to the realm of computer programs.
- In 1978 he developed a theory *Communicating Sequential Processes* (CSP) — a formal language to describe concurrent systems, which included the concept of *monitors*; both have a foundational importance for the theory of parallel computations.
- In 1965 he made a Billion Dollar Mistake ("simply because it was so easy to implement").

NULL Reference

Tony Hoare (2009)

> *I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.*

Conversations-1

With OO-Hack of NULL-pointer

- Hello, is this IT support?
- *Yes.*
- Let me talk to your employee “Bruce”, please.
- *Hold the line please... Hello!*
- Are you `Nobody`?
    - [People in the same room] What a strange person!

Conversations-2

With Exceptions

- Hello, is this IT support?
- *Yes.*
- Let me talk to your employee "Bruce", please.
- *beep-beep-beep*
- Hey boss! They hung up!
  1. [Boss] Do it yourself, then!
  2. [Boss] Call an independent contractor!

Conversations-3

With Optional

- Hello, is this IT support?
- *Yes.*
- Let me talk to your employee "Bruce", please.
- *This is Michael. Can* I *help you?*
- [caller response]
  1. Ok, maybe you *can*... Michael, you are really Bruce, are you not?
  2. No, thanks, I'll tell my supervisor that Bruce isn't available, we'll think of something else. Cheers.

Optional

`java.util.Optional<T>` is a wrapper to either represent an object of type `T`, or no object (which would be `null`). *Optional* helps to deal safely with the situation when a value *can be* `null`.

- When the object is there, `isPresent() -> true`
- Get a regular value, or (if `null`) a default one:

```
String result = optionalString.orElse(""); // empty if null inside
```

- Get a regular value, or (if `null`) calculate one:

```
String result = optionalString.orElseGet(() -> System.getProperty("user.dir"));
```

- Get a regular value, or (if `null`) throw an exception:

```
String result = optionalString.orElseThrow(IllegalStateException::new);
```

- If a regular value, it can be used in a computation, or (if `null`) nothing happens

```
optionalValue.ifPresent(v -> Process v); // but no value is returned
```

- `Optional.get()` returns a regular value, or throws `NoSuchElementException` (therefore, do not use it as a extraction value trick — it looses all safety of *Optional*)
- To create an *Optional* object, use *static* methods `of(t)` and `empty()`

```
public static Optional<Double> inverse(Double x) {
   return x == 0 ? Optional.empty() : Optional.of(1/x);
}
```

Further Reading

- Hortsmann *Core Java for the Impatient*, Ch. 5.1, 5.2, 8.7.1–8.7.3
- Oracle *The Java Tutorials*: Exceptions
- Raoul-Gabriel Urma Tired of Null Pointer Exceptions? Consider Using Java SE 8's Optional!