Exceptions

CSC207 Winter 2015

Computer Science
UNIVERSITY OF TORONTO

What are exceptions?

Exceptions report exceptional conditions: unusual, strange, disturbing.

These conditions deserve exceptional treatment: not the usual go-to-the-next-step, plod-onwards approach.

Therefore, understanding exceptions requires thinking about a different model of program execution.

Exceptions in Java

To **"throw an exception"**:

```
throw Throwable;
```

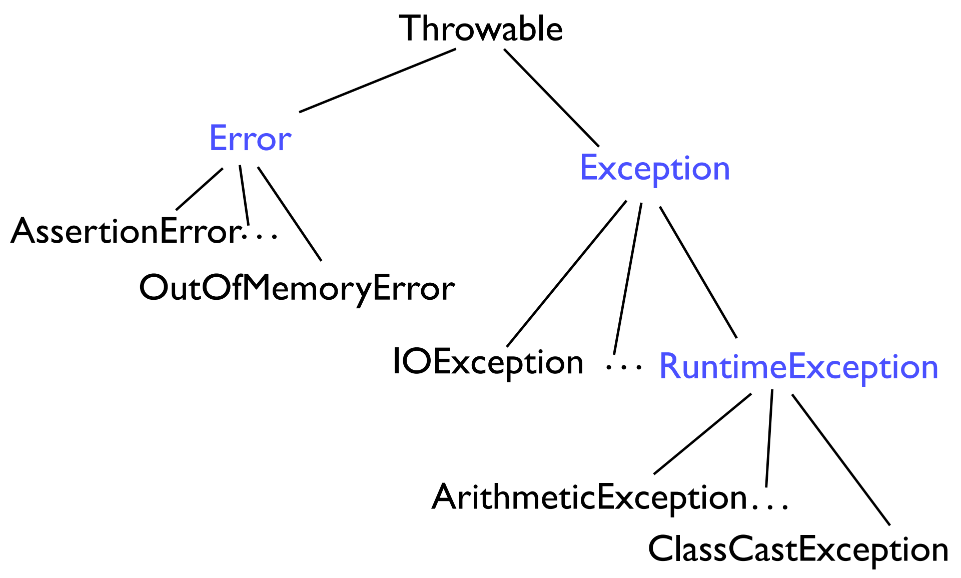
To **"catch an exception"** and deal with it:

```
try {
    statements
// The catch belongs to the try.
} catch (Throwable parameter) {
    statements
}
```

To say you aren't going to deal with exceptions (or may throw your own):

```
accessMod returnType methodName (parameters)
throws Throwable { ... }
```

The Hierarchy



`Throwable`

Constructors:

`Throwable(), Throwable(String message)`

Other useful methods:

`getMessage()`

`printStackTrace()`

`getStackTrace()`

What should you `throw`?

You can throw an instance of `Throwable` or any subclass of it (whether an already defined subclass, or a subclass you define).

Don’t throw an instance of `Error` or any subclass of it: these are for unrecoverable circumstances. (e.g., `OutOfMemoryError`).

Don’t throw an instance of `Exception`: throw something more specific.

It’s okay to throw instances of:

- specific subclasses of `Exception` that are already defined, e.g., `UnsupportedOperationException`
- specific subclasses of `Exception` that you define.

These do not need to be handled

`Error`:

“Indicates serious problems that a reasonable application should not try to catch.”

Do not have to handle these errors because they “are abnormal conditions that should never occur.”

`RuntimeException`:

These are called **unchecked** because you do not have to handle them.

Checked vs. Unchecked Exceptions

When defining an `Exception` subclass, we need to decide whether to extend `RunTimeException` (unchecked) or `Exception` (checked).

```
public class MyException extends RuntimeException {...}
class MyClass {
    public void m()  /* No "throws”, but it compiles! */  {...
        if (...) throw new MyException("oops!") {...}
  }
}

public class MyException extends Exception {...}
class MyClass {
    public void m() throws MyException { ...
    if (...) throw new MyException("oops!") {...}
  }
}
```

What does the Java API say?

`Exception`:

“The class `Exception` and its subclasses are a form of `Throwable` that indicates conditions that a reasonable application might want to catch.”

`RunTimeException` (unchecked):

“`RuntimeException` is the superclass of those exceptions that can be thrown during the normal operation of the Java Virtual Machine.”

Examples: `ArithmeticException`, `IndexOutOfBoundsException`, `NoSuchElementException`, `NullPointerException`

non-`RuntimeException` (checked):

Examples: `IOException`, `NoSuchMethodException`

Guideline for which to use

"Use **checked exceptions** for conditions from which the caller can reasonably be expected to recover."

"Avoid unnecessary use of **checked exceptions**."

If the user didn’t use the API properly or if there is nothing to be done, then make it a RunTimeException.

"Use **run-time exceptions** to indicate programming errors. The great majority of run-time exceptions indicate precondition violations."

Example: Suppose method getItem(int i) returns an item at a particular index in a collection and requires that i be in some valid range.

The programmer can check that before they call o.getItem(x).

So sending an invalid index should not cause a checked exception to be thrown.

We can have cascading catches

Much like an if with a series of else if clauses, a try can have a series of catch clauses.

After the last catch clause, you can have a clause:

```
finally { ... }
```

But finally is not like a last else on an if statement:
The finally clause is always executed, whether an exception was thrown or not, and whether or not the thrown exception was caught.

Example of a good use for this: close open files as a clean-up step.

An example of multiple catches

Suppose `ExSup` is the parent of `ExSubA` and `ExSubB`.

```
try {
   ...
} catch (ExSubA e) {
   // We do this if an ExSubA is thrown.
} catch (ExSup e) {
   // We do this if any ExSup that's not an ExSubA is thrown.
} catch (ExSubB e) {
   // We never do this, even if an ExSubB is thrown.
} finally {
   // We always do this, even if no exception is thrown.
}
```

`finally` vs. code after `try/catch`

```
try {
  // do something
} catch(MyException e) {
  // handle exception
} finally {
  cleanUp();
}
```

```
try {
  // do something
} catch(MyException e) {
  // handle exception
}
cleanUp();
```

Even if there are `return` statements or exceptions in the `try` or `catch` blocks, the code in `finally` be executed. That isn't the case with the code on the right-hand side.

Documenting Exceptions

```
/**
 * Return the mness of this object up to mlimit.
 * @param mlimit The max mity to be checked.
 * @return int The mness up to mlimit.
 * @throws MyException If the local alphabet has no m.
 */
public void m(int mlimit) throws MyException { ...
    if (...) throw new MyException ("oops!") { ...
    }
}
```

You need both:

the Javadoc comment is for human readers, and

the `throws` is for the compiler.

Both the reader and the compiler are checking that caller and callee have consistent interfaces.