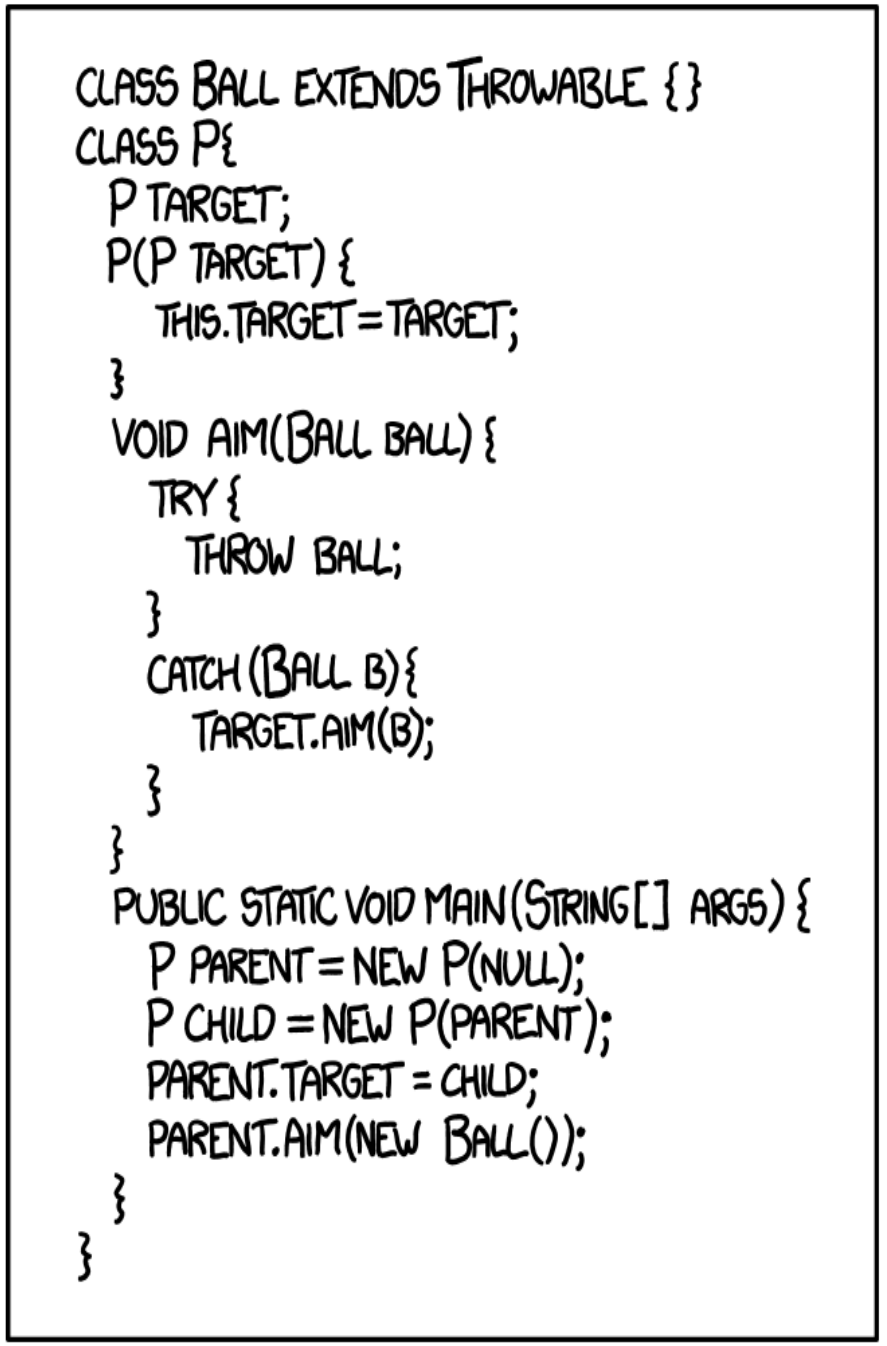
EXCEPTIONS IN JAVA

CSC 207 SOFTWARE DESIGN

BONDING BY XKCD

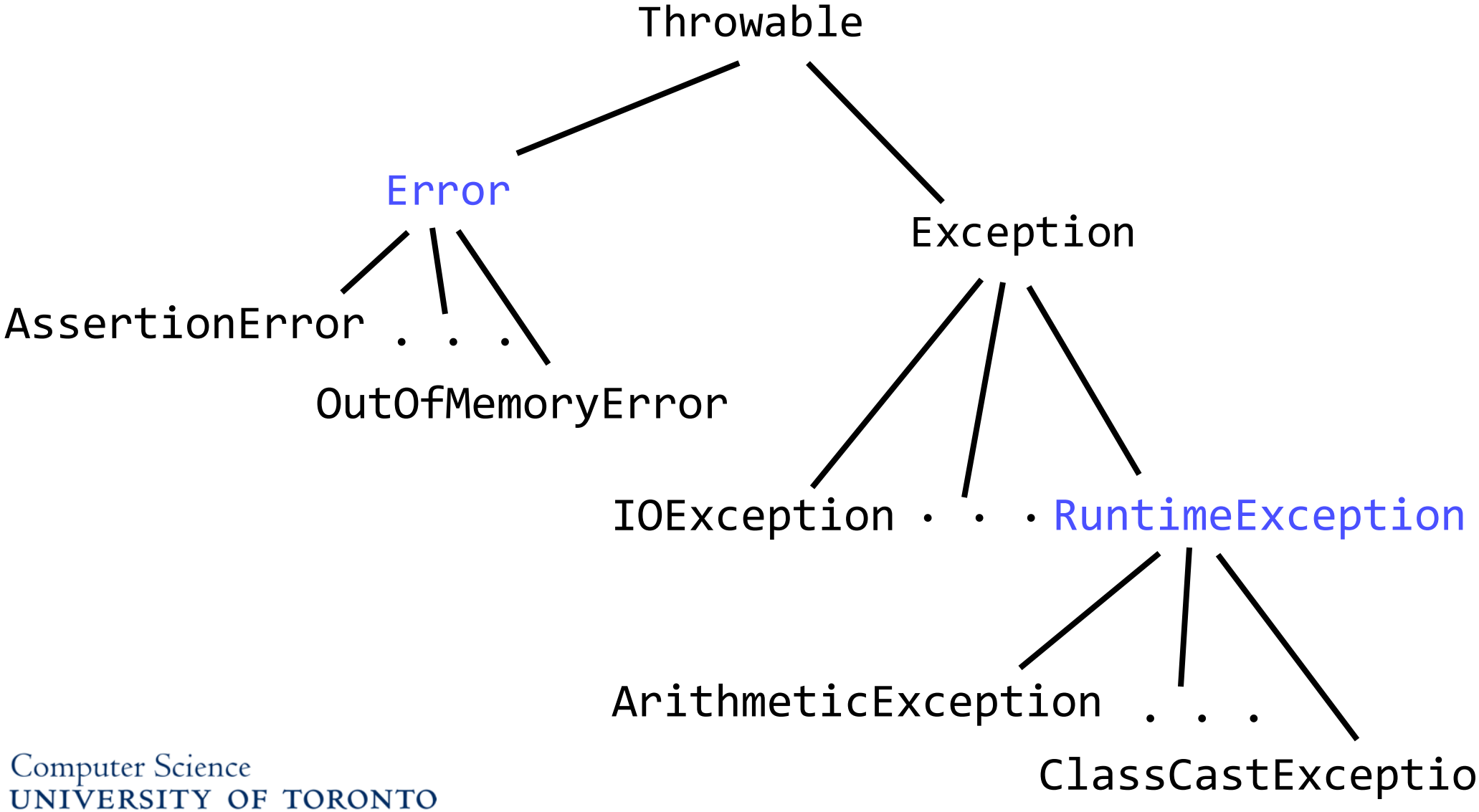
https://github.com/paulgries/XXCDBall.git



LEARNING OUTCOMES

- Understand how Exceptions work in Java
- Be familiar with the Throwable hierarchy
- Know the difference between checked and unchecked exceptions — and when to use each of them

THE THROWABLE HIERARCHY



WHAT ARE EXCEPTIONS?

- Exceptions report **exceptional conditions**: unusual, strange, unexpected.

- 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.

SOME METHODS THROW EXCEPTIONS

```
public boolean addAll(int index,
                      Collection<? extends E> c)
```

Inserts all of the elements in the specified collection into this list, starting at the specified position. Shifts the element currently at tl

Throws:

`IndexOutOfBoundsException` - if the index is out of range (`index < 0 || index > size()`)

`NullPointerException` - if the specified collection is null

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/ArrayList.html#add(int,E)

EXCEPTIONS IN JAVA

- To “throw an exception”:

  `throw new` *Throwable(message)*`;`

- To “catch an exception” and deal with it:

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

- To say a method isn’t going to deal with exceptions (or may throw its own):

  accessModifier returnType methodName(*parameters*) **throws *Throwable*** { ... }

ANALOGY

- `throw`
  - I'm in trouble, so I throw a rock through a window, with a message tied to it. It's like a return statement (exits the current method) but bad.
- `try`
  - Someone in the following block of code might throw rocks of various kinds. All you catchers line up ready for them.
- `catch`
  - If a rock of my kind comes by, I'll catch it and deal with it.
- `throws`
  - I'm warning you: if there's trouble, I may throw a rock.
- Even though there are only two new statement types, this changes the whole picture of how a program runs.

(AWKWARD) EXAMPLE

```
int i = 0;
int sum = 0;
try {
  while (true) {
    sum += i++;
    if (i >= 10) {
      // we're done
      throw new Exception("i at limit");
    }
  }
} catch (Exception e) {
  System.out.println("sum to 10 = " + sum);
}
```

WHY WAS THAT CODE BAD?

- The situation that the exception reports is not exceptional.
  - It’s obvious that `i` will eventually be 10. It's expected.
  - In Java, exceptions are reserved for exceptional situations.
- It's uncharacteristic. Real uses of exceptions aren't local.
  - `throw` and `catch` aren't generally in the same block of code.

WHY USE EXCEPTIONS?

- Less programmer time spent on handling errors
- Cleaner program structure:
  - isolates exceptional situations rather than sprinkling them through the code
- Separation of concerns:
  - Pay local attention to the algorithm being implemented and global attention to errors that are raised

WE CAN HAVE CASCADING 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 is 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!

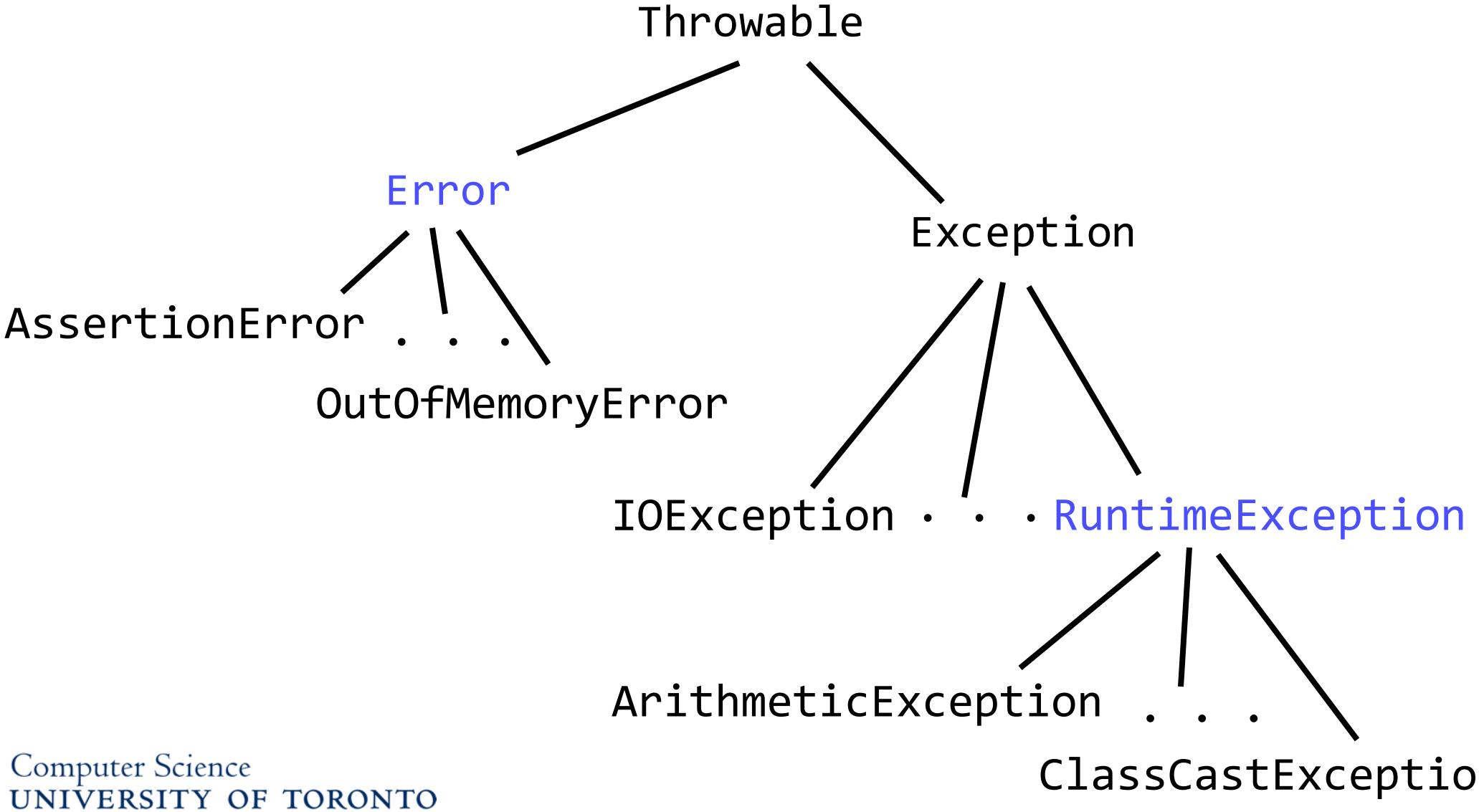
- After the last `catch` clause, you can have a clause:
  - `finally { ... }`

- 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.

RECAP

- If you call code that may throw a non-Runtime exception, you have two choices:
  - you can wrap the code in a `try-catch`, or
  - you can use "`throws`" to declare that your code may throw an exception.
- Exceptions don't follow the normal control flow.
- Some guidelines on using exceptions well:
    - Use exceptions for exceptional circumstances.
    - Throwing and catching should not be in the same method. "Throw low, catch high".

THE THROWABLE HIERARCHY



“THROWABLE” HAS USEFUL METHODS

- Constructors:
  - `Throwable()`, `Throwable(String message)`
- Other useful methods:
  - `getMessage()`
  - `printStackTrace()`
  - `getStackTrace()`
- You can also record (and look up) within a `Throwable` it’s “cause”: another `Throwable` that caused it to be thrown. Through this, you can record (and look up) a chain of exceptions.

CHECKED VS UNCHECKED EXCEPTIONS

- Checked Exceptions:
    - A method **must** include "throws ExceptionClassName" in its declaration
    - The compiler will check to make sure that the Exception is caught by the method that calls it, or the method that calls that method, or...
- Unchecked Exception:
    - The program will throw these without checking to see if it is eventually caught
    - Will crash the program and print the stack trace to the terminal
    - You can then see where the Exception was thrown and fix the code so that does not happen again (helpful when debugging!)

YOU DON'T HAVE TO HANDLE ERRORS OR RUNTIME EXCEPTIONS

- Error:
  - "Indicates serious problems that a reasonable application should not try to catch."
  - The programmer does not have to handle these errors because they "are abnormal conditions that should never occur."
- RuntimeException:
  - These are called "unchecked" because you are not required to catch them.
  - These typically indicate that something bad and unexpected happened, such as not being able to read from a text file because of a permission problem, or not being able to send a message because a network connection failed.
  - These are typically handled at one place in your program, and details are logged so that the programmers can investigate.

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.
- **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.

EXTENDING EXCEPTION: VERSION 1

Example: a method `m()` that throws your own exception `MyException`, a subclass of `Exception`:

```
class MyException extends Exception {...}

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

EXTENDING EXCEPTION: VERSION 2

Example: a method `m()` that throws your own exception `MyException`, a subclass of `Exception`:

```
class MyClass {
  public static class MyException extends Exception {...}

  public void m() throws MyException { ...
    if (...) throw new MyException("oops!"); ...
  }
}
```

ASIDE: CLASSES INSIDE OTHER CLASSES

- You can define a class inside another class. There are two kinds.
- Static nested classes use keyword `static`.
  (It can only be used with classes that are nested.)
    - Cannot access any other members of the enclosing class.
- Inner classes do not use keyword `static`.
    - Can access all members of the enclosing class
      (even private ones).
- Nested classes increase encapsulation. They make sense if you won't need to use the class outside its enclosing class.
- Reference:
  http://docs.oracle.com/javase/tutorial/java/javaOO/nested.html

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!"); ...
}
```

- the Javadoc comment is for human readers, and
- keyword `throws` is for the compiler.
- Both the reader and the compiler are checking that caller and callee have consistent interfaces.

EXTEND EXCEPTION OR RUNTIME EXCEPTION?

Recall that RuntimeExceptions are not checked. Example:

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

- How do you choose whether to extend `Exception` or `RuntimeException`?
- Perhaps you should always extend `Exception` to benefit from the compiler's exception checking?

WHAT DOES THE JAVA API SAY?

- `Exception`
  - "Class `Exception` and its subclasses are a form of `Throwable` that indicate conditions that a reasonable application might want to catch."
- `RuntimeException` (not checked)
  - "RuntimeException is the superclass of those exceptions that can be thrown during the normal operation of the Java Virtual Machine."
  - Example subclasses: `ArithmeticException`, `IndexOutOfBoundsException`, `NoSuchElementException`, `NullPointerException`
- non-`RuntimeException` (checked)
  - Example subclasses: `IOException`, `NoSuchMethodException`

WHAT DOES THE JAVA LANGUAGE SPECIFICATION SAY?

"The runtime exception classes (`RuntimeException` and its subclasses) are exempted from compile-time checking because, in the judgment of the designers of the Java programming language, having to declare such exceptions **would not aid significantly in establishing the correctness** of programs."

"The information available to a compiler, and the level of analysis the compiler performs, are usually not sufficient to establish that such run-time exceptions cannot occur, even though this may be obvious to the programmer. Requiring such exception classes to be declared **would simply be an irritation to programmers**."

https://docs.oracle.com/javase/specs/jls/se7/html/jls-11.html

EXAMPLE

- Imagine code that implements a circular linked list.
- Suppose that, by construction, this structure can never involve null references.
- The programmer can then be certain that a `NullPointerException` cannot occur.
- But it would be difficult for a compiler to prove it!
- So if `NullPointerException` were checked, the programmer would have to handle them (catch or declare that one might be thrown) ***everywhere*** a `NullPointerException` might occur.

ADVICE FROM JOSHUA BLOCH

HTTPS://EN.WIKIPEDIA.ORG/WIKI/JOSHUA_BLOCH

- "Use checked exceptions for conditions from which the caller can reasonably be expected to recover."
- "Use run-time exceptions to indicate programming errors. The great majority of run-time exceptions indicate precondition violations."
  - **if the programmer could have predicted the exception, don't make it checked.**
  - 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 range *before* they call `o.getItem(i)`.
    - Passing an invalid index should not cause a checked exception to be thrown in this case.
- "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.

SOLID AND EXCEPTIONS

Exceptions in Java are well designed.

Agree or disagree? Justify your answer.

SOLID AND EXCEPTIONS

SRP?

Yes!

- Allows separation of normal return conditions and exceptional situations, so programmers aren’t trying to deal with both normal and exceptional cases using the return mechanism.

- Exception classes are created in the context of a problem specification. They are used to indicate violates of the core business logic of a system. The "actor" is the person responsible for deciding on that core business logic. There is no behaviour (so no methods) in exceptions.

SOLID AND EXCEPTIONS

OCP?

Yes!

- We can extend Exception or RuntimeException to enrich how our programs handle various exceptional situations. They really are designed to be extended!
- Since we throw Exception objects, inheritance also applies to exceptions. So a catch block that originally caught one subtype of Exception will be able to catch subsequent subtypes if we decide to make our exceptions more specific by extending the ones we were already using.
- This follows the Open/Closed Principle because it means we can add new subtypes of Exception and throw them in the place of their superclasses without having to change our catch blocks.

SOLID AND EXCEPTIONS

LSP?

Can we replace instances of an exception with instances of a subclass without causing the program to break?

Yes!

We are required (using the throws keyword) to declare when we throw checked exceptions, but not when we may throw unchecked exceptions (we still can declare throws, but it isn’t required). Having checked exceptions above unchecked in the hierarchy is consistent with LSP. The opposite would not be.

A subclass that overrides a method which throws a checked exception is only allowed to throw exceptions which are unchecked or instanceof the original exception that could be thrown in the parent method. You can't throw a plain Exception, or any other non-Runtime exception, in an overriding method, because the throws clause is part of the method signature. The code won't compile. This prevents us from introducing Liskov violations.

SOLID AND EXCEPTIONS

ISP?

Yes!

The public interface of Throwable is focused and, since it is a class, we have little to nothing to implement ourselves when subclassing an Exception or RuntimeException.

Exceptions declared to be thrown by methods in an interface should be relevant to all its potential implementers, so that the implementing classes don't need code to handle irrelevant exceptions.

SOLID AND EXCEPTIONS

DIP?

…maybe-ish?

We don’t have any abstractions, so it doesn’t seem to be directly relevant…

WHO IS THE EXCEPTION FOR?

- throwable can also contain a **cause**: another throwable that caused this throwable to be constructed. The recording of this causal information is referred to as the chained exception facility, as the cause can, itself, have a cause, and so on, leading to a "chain" of exceptions, each caused by another.

- One reason that a throwable may have a cause is that the class that throws it is built atop a lower layered abstraction, and an operation on the upper layer fails due to a failure in the lower layer. **It would be bad design to let the throwable thrown by the lower layer propagate outward, as it is generally unrelated to the abstraction provided by the upper layer.** Further, doing so would tie the API of the upper layer to the details of its implementation, assuming the lower layer's exception was a checked exception. Throwing a "wrapped exception" (i. e., an exception containing a cause) allows the upper layer to communicate the details of the failure to its caller without incurring either of these shortcomings. It preserves the flexibility to change the implementation of the upper layer without changing its API (in particular, the set of exceptions thrown by its methods).

-- Javadoc for the Throwable class

DIP EXAMPLE

- A class that interacts with a SQL database (you'll learn about this in CSC343) may have to deal with an SQLException. There are other kinds of databases, though, and client code may choose different databases to read from.

- That client code should not have to deal with SQLExceptions, since the database being SQL-based is a detail of the implementation! Instead, the program's DatabaseInterface will define a DataAccessException class, and the SQL code should wrap the SQLException in a DataAccessException:

```
 try { doSomeSQLDatabaseStuff()
} catch (SQLException e) {

 // Wrapping the low-level exception in a higher-level
custom exception

    throw new DataAccessException("Failed to save
user to the database.", e);

}
```

- This lets the client code work abstractly:

```
public void registerUser(User user, DatabaseInterface
myDatabase) {
  try {
      myDatabase.saveUser(user);
  } catch (DataAccessException e) {
   // Handle the error in a way relevant to the service
layer
        System.out.println("Error: Unable to register user.
Please try again later.");
// You might log the exception details or take other
action as needed
  }
}
```

```
interface DatabaseInterface {
   SaveUser(User user) throws DataAccessException
}
```