Learning Objectives

- Parametric Polymorphism (and Generics)
- Error Handling

Parametric Polymorphism

We've discussed **polymorphism** of objects. **Parametric polymorphism** relates to methods. It allows parameters to be bound to a diversity of object types.

- Polymorphism is realized in java via **generic** types.
- They were introduced as in Meta Language (ML) in **1976.**
- We see them frequently used with **Collections**.

https://en.wikipedia.org/wiki/ML_(programming_language)

Parametric Polymorphism Example

You've seen syntax that looks something like this:

```
List<Type> name = new ArrayList<Type>();
```

The <**Type**> here indicates the type of object that we expect our ArrayList to hold. It is a **parameter** that we pass to the interface or class, in order to specify a **generic** type.

We can parameterize classes to accept generic types using syntax that looks like this:

```
public class name<Type> {}
```

This tells clients they need to indicate a **type parameter** when **constructing objects** of the type.

Parametric Polymorphism

There are categories of **generic type,** e.g.

```
public class Name<E> {}
```

The convention is to use a 1-letter names for the categories:

T for Type, E for Element, N for Number, K for Key, or V for Value.

The specific type of the parameter is provided when class objects are instantiated, e.g.

```
Name<String> myObject = new Name<String>();
```

This says **E** will be cast as a **String** within the object's methods.

Parametric Polymorphism

```
public class Name<E> {
    ...
    public static int method(E value) {
        for (int i = 0; i < size; i++) {
            if (elementData[i].equals(value)) {
                return I;
            }
        }
        return -1;
    }
}
```

The compiler will know to relate the generic **E** to the **E** in class methods because of the class declaration above. We can use a different syntax with non-class methods:

```
public class Name {
    ...
    public static <E> int method(E value) {
        for (int i = 0; i < size; i++) {
            if (elementData[i].equals(value)) {
                return I;
            }
        }
        return -1;
    }
}
```

In the code above, we use **equals** and not **"=="** *(why not?).*

Upper Bounds on Generics

```
public class TreeSet<T extends Comparable<T>> {
    ...
}
```

A **generic type** can be **extended**.

The code above tells us that a TreeSet can be constructed from any **comparable type of object** or any **subtype** of a comparable object.

Comparable<T> therefore serves as an **upper bound on the type of T.**

Wildcards

? indicates a **Wildcard parameter** and can be of any type, e.g.

```
List<?> list = new List<?>();
```

Wildcards are special in part because they support both **upper and lower bounds on typing** e.g.

```
public class ObjectName<? super Animal> {
    ...
}
```

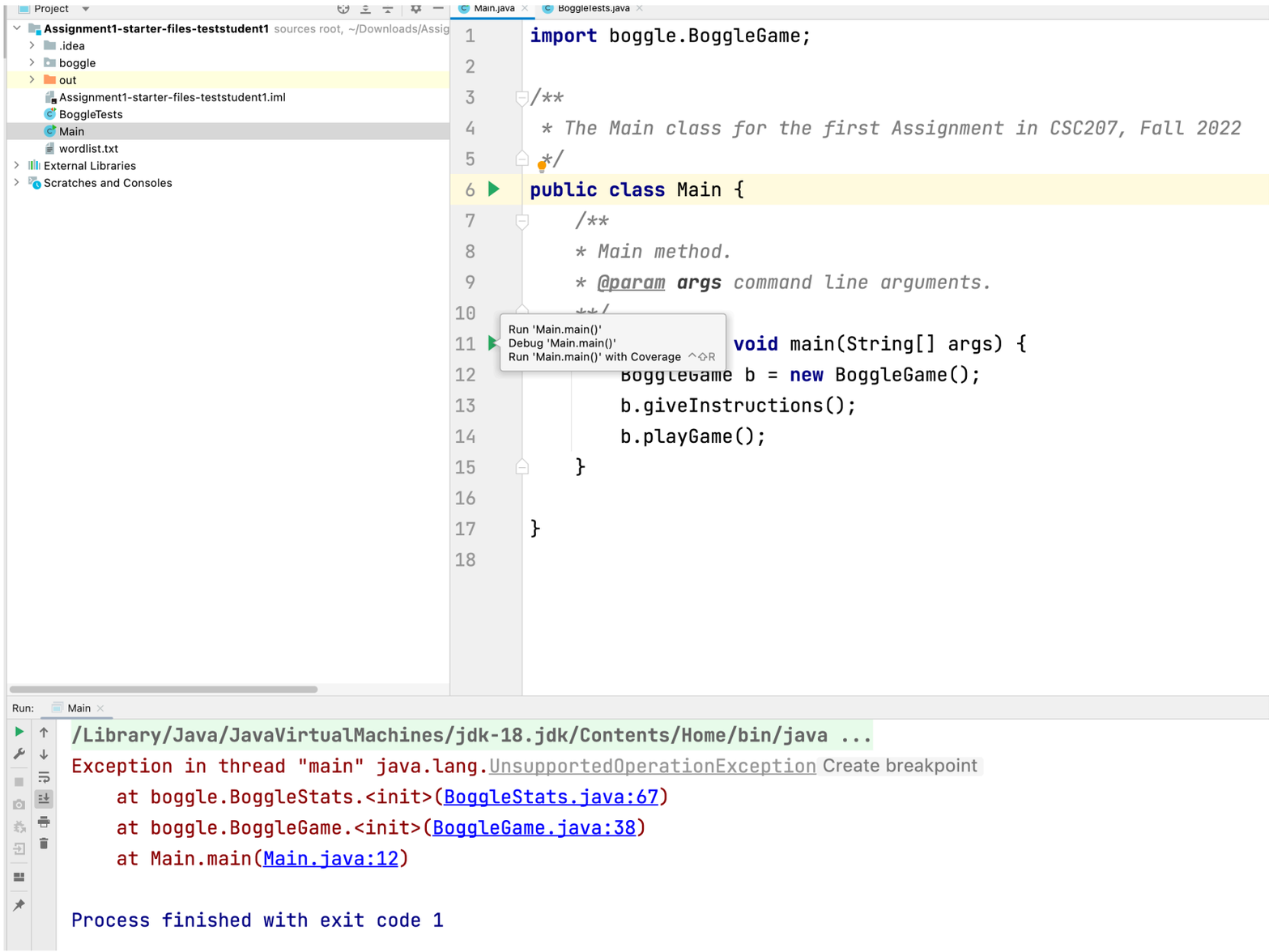
This says an **ObjectName** can be constructed from any **Animal** or any **superclass** of an **Animal**. **Animal** therefore serves as a **lower bound on the Object's type.**

More Information About Generics

For more information about generics:

https://docs.oracle.com/javase/tutorial/java/generics/types.html

Exceptions: Motivating Examples



We've now seen some "Exceptions" in our Labs.

In particular, we've seen exceptions that tells us some operation has yet to exist.

UNIVERSITY OF TORONTO MISSISSAUGA

Exception: Motivating Example

```
public class Kaboom {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        List<String> myList = new ArrayList<>();
        myList.get(1).length(); // kaboom
    }
}
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index 1 out of
bounds for length 0
    at java.base/
jdk.internal.util.Preconditions.outOfBounds(Preconditions.java:64)
    at java.base/
jdk.internal.util.Preconditions.outOfBoundsCheckIndex(Preconditions.java:70)
    at java.base/
jdk.internal.util.Preconditions.checkIndex(Preconditions.java:266)
    at java.base/java.util.Objects.checkIndex(Objects.java:359)
    at java.base/java.util.ArrayList.get(ArrayList.java:427)
    at exceptions.Kaboom.main(Kaboom.java:10)
```

You may have encountered other “Exceptions” along the way, too.

But what are these exceptions, and how can we manage them with grace?

UNIVERSITY OF TORONTO MISSISSAUGA

Exceptions Defined

Exceptions report exceptional conditions.

An UnsupportedOperation Exception tells as an operation doesn't exist:

```
throw new UnsupportedOperationException(); //replace this line!!
```

This exception immediately halts the method that is executing and continues halting methods down the call stack until:

- one of the methods **deal** withs the **problem** (or **catch** the **exception**), after which the program continues running normally, or

- the entire call stack is empty, at which point the user sees a message printed to the console about the exception.

An Exception is an **object** of some **Exception class.**

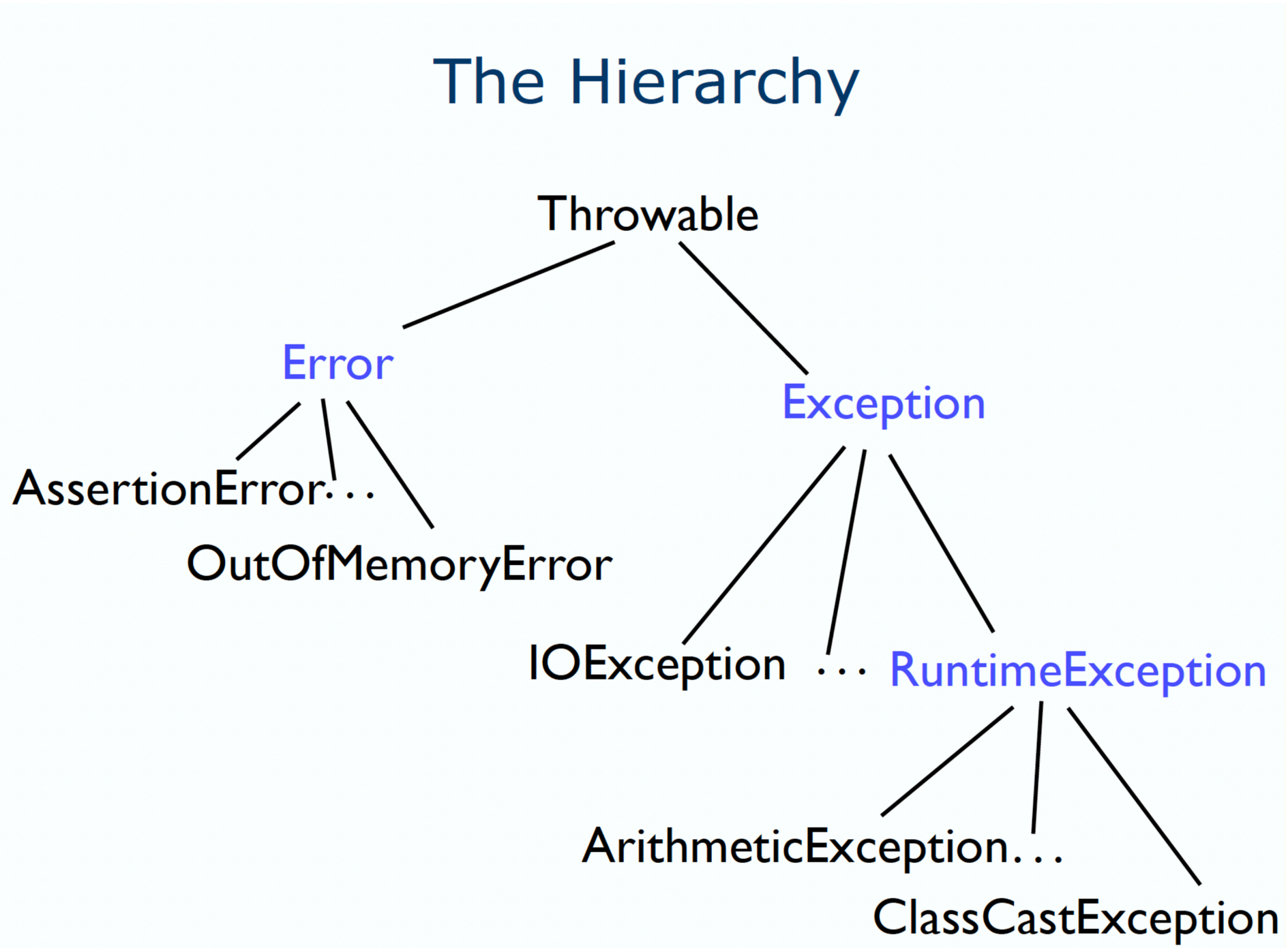
Why Exceptions?

Why not just use special return values? These can be checked and handled, like Exceptions.

- This approach would require sending and checking special values all the way down the call stack.
- Contradicts the **Single Responsibility Principle!**

With Exceptions, error-handling code can be in one location within the call stack, which promotes **modularity** and **code reuse.**

Exceptions are “Throwable” Objects



https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/lang/Throwable.html

Throwable Objects

Constructors: **Throwable(), Throwable(String message)**

Other useful methods:

getMessage()

printStackTrace()

getStackTrace()

You can also record (and look up) a Throwable's "cause", i.e. the Throwable method that caused it to be thrown. In this way you can record (and look up) a **chain** of **Exceptions**.

Throwable Objects: SubTypes

Error:

- Indicates serious problems that an application **should not try to catch.**

- Developers do not have to handle these errors because they ***are abnormal conditions that should never occur.*** If this kind of Exception happens, it's on those that are using our code.

Exception:

- The class **Exception** and its subclasses are forms of Throwable that indicate conditions that an application might want to catch. These are exceptions that you, and your developers, may want to handle.

UNIVERSITY OF TORONTO MISSISSAUGA

Unchecked Throwable Objects

RuntimeException:

Is a **subclass** of an Exception that includes exceptions that can be thrown during the normal operation of the Java Virtual Machine.

Examples: ArithmeticException, IndexOutOfBoundsException, NoSuchElementException, NullPointerException

These are called ***unchecked*** because Java does not require you to declare that you will throw or handle them.

i.e., for a RuntimeException, it's okay if you don't catch it and you don't declare that you might throw it, either.

Checked Throwable Objects

Exceptions that are NOT a RuntimeException are **checked**

They include: **IOException, NoSuchMethodException** and those that you invent.

These are exceptions that you must declare to throw and/or handle

Which to Use and When

Use run-time (unchecked) exceptions to indicate programming errors (e.g. precondition violations).

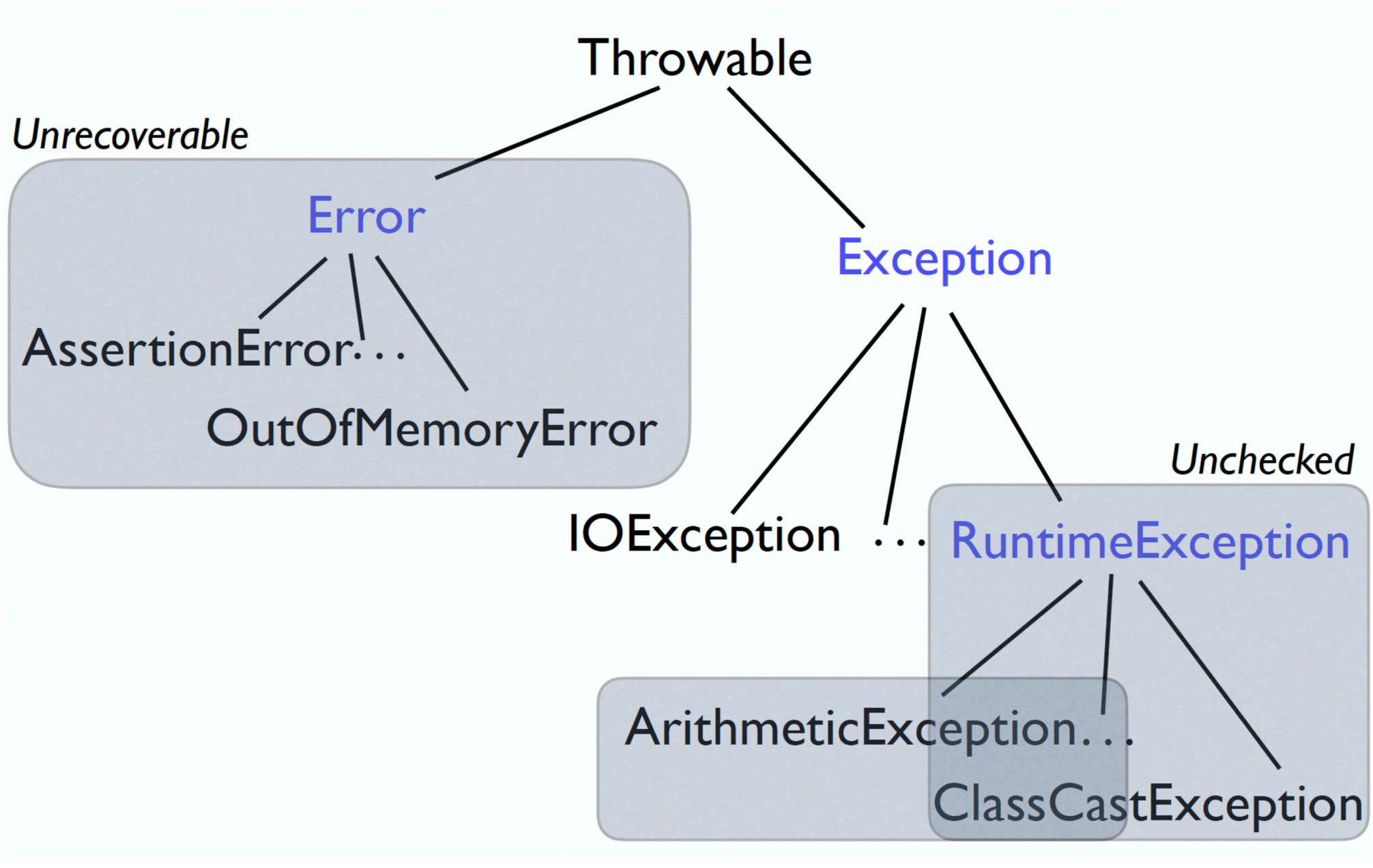
If the programmer could have predicted the exception, don't make it checked.

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.

If preconditions are documented, whomever uses this method should check for the error before they call o.getItem(x).

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

In Summary



Using Exceptions: Syntax

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
}
```

Your methods may “throw” a Throwable up the call stack:

```
public void methodName (parameters) throws Throwable { .. }
```

UNIVERSITY OF TORONTO MISSISSAUGA

The important keywords

throw:

I'm in trouble so I am throwing a rock through a window with a message tied to it.

try:

The next block of code might throw rocks of various kinds. All you “catchers” …. line up and be ready for them.

catch:

If a rock of my kind comes by, I'll catch it and deal with it.

throws:

Warning: if there’s trouble, I am a class that may throw a rock.

The finally clause

```
public class MoreKaboom {

    public static void main(String[] args) {
        int numerator, denominator;
        Scanner reader = new Scanner(System.in);
        System.out.println("Numerator, please:");
        numerator = reader.nextInt();
        System.out.println("Denominator, please:");
        denominator = reader.nextInt();
        try {
            int quotient = (int)numerator/denominator;
            System.out.println("Quotient of "+numerator+
                    " divided by "+denominator+" is "+quotient);
        } catch (ArithmeticException e) {
            System.out.println("Division by zero!");
        } finally {
            System.out.println("I tried to divide, I hope I succeeded!");
        }
    }
}
```

Finally clauses will be executed regardless of the exception.

UNIVERSITY OF TORONTO MISSISSAUGA

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'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.
}
```

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.

Learning Objectives

- Requirements Lifecycles
- User Stories

What is the role of requirements?

- Identify customers' needs.
- Assess feasibility, technical requirements and costs.
- Design the purpose of system components.
- Establish schedules and discover constraints.

Lifecycle of requirements

- **Requirements Elicitation:** work with customer to assess what the software should do, how it will fit into their workflow or lifestyle, how it will support their objectives.

- **Requirements Analysis:** Organize requirements into related or co-dependent groups, review them for correctness, prioritize them based on customers' input.

- **Requirements Specification:** Translate requirements into tasks or specifications that developers understand and which detail function, development constraints, and expected performance.

- **Requirements Validation:** Ensure requirements conform to customer's expectations, development standards, stakeholder interests.

Lifecycle of requirements

- **Requirements Elicitation:** work with customer to assess what the software should do, how it will fit into their workflow or lifestyle, how it will support their objectives.

- **Requirements Analysis:** Organize requirements into related or co-dependent groups, review them for correctness, prioritize them based on customers' input.

- **Requirements Specification:** Translate requirements into tasks or specifications that developers understand and which detail function, development constraints, and expected performance.

- **Requirements Validation:** Ensure requirements conform to customer's expectations, development standards, stakeholder interests.

Requirement Elicitation

- Meet with your clients!
- Ask:
  - About other stakeholders that may be involved
  - About their goals for the system
  - About the context in which the system will be used
  - About their feelings regarding proposed solutions
  - About their feelings as to your progress
- Talk to many stakeholders, if many will be involved in decision making
- Talk to many customers, if you expect many customers to use your system
- Write initial definitions of requirements in language that customers can understand

User Stories

What are they?

- Descriptions of software sensitive to customer/user perspective
- Precise, compact, parsimonious, limited in detail
- Describes a specific software features in non-technical jargon
- Can ideally be validated by stakeholders/customers/users
- Facilitates estimates of development time, scheduling, resource allocation
- Can be associated with **acceptance tests,** to verify that a story has been properly implemented

User Stories

What are they?

- Follow a syntax:

“As a [persona], I [want to], [so that].”

User Story:

As a bank customer, I want to Login to my account using a bank card and PIN code so that I can perform a transaction.

Acceptance Tests:

Correctly validate bank card and PIN code
Lock card if user enters incorrect PIN 3+ times

Ambiguous Words

acceptable, adequate
as much as practicable
at least, at minimum, not more than, not to exceed
between
depends on
efficient
user-friendly, simple, easy
flexible
improved, better, faster, superior
including, including but not limited to, and so on, such as
maximize, minimize, optimize
normally, ideally
optionally
reasonable, when necessary, where appropriate

OTHERS?

UNIVERSITY OF TORONTO MISSISSAUGA