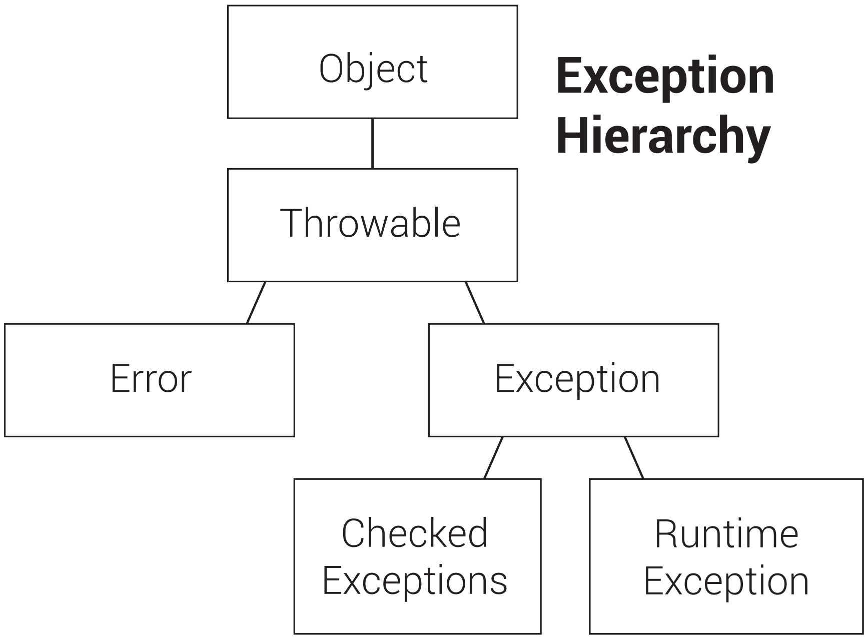
CSM Week 3 Review

Exceptions

`Throwable` is a superclass with two subclasses, **`Error`** and **`Exception`**. **`Error`** objects indicate serious problems with your application and a reasonable application should not catch one. They are in a separate class from **`Exception`** because they cannot be handled. An example is **`OutOfMemoryError`**, which cannot be fixed with programming. At this point, we want our program to end to fix it. Also, any exceptions thrown under the **`Error`** class and its subclasses are considered unchecked exceptions. Don't worry too much about **`Error`**! It is out of scope.



Exception Hierarchy

`Exception` objects, on the other hand, can be caught and handled in with programming. There are two types of exceptions:

Checked Exceptions— exceptions that need to be explicitly handled and are caught by the compiler

Examples: file name is not equal to class name, casting an object to an independent class

Unchecked Exceptions— exceptions that are not caught by the compiler, but thrown during runtime

Examples: NullPointerException, ArrayIndexOutOfBoundsException, ClassCastException

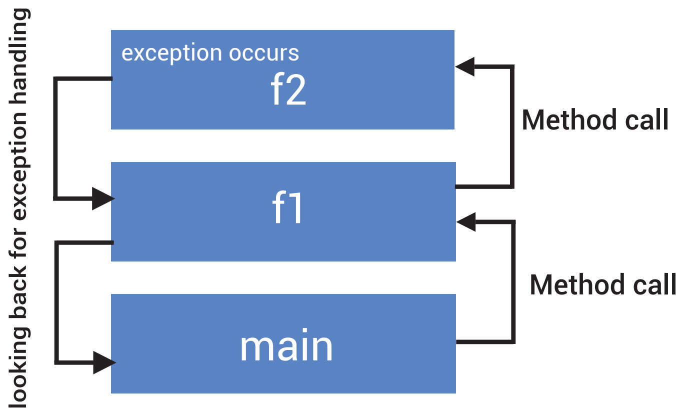
Note: These are exception subclasses that explicitly extend the RuntimeException class.

Why are exception objects useful?

We want to be able to catch and handle exceptions during runtime. This makes debugging easier. Sometimes, programs want to do something with exceptions. Suppose we're programming a game. If we have any conditions that satisfy game over, we can purposely throw a **`RuntimeException`**, print out 'Game Over', and immediately end the game.

What actually happens if an exception occurs?

Suppose we have a main method that calls a **`f1`** method, which calls a **`f2`** method and an exception occurs. This creates an *exception object* that must be handled by the runtime system. What will happen is the runtime system will go down the call stack searching for something that will handle the exception. If it finds nothing, the program will terminate with the exception. If a **`try-catch block`** catches the exception, then the exception is considered *handled.*



Call Stack Propogation

Try-Catch Block

Try-catch blocks in Java are very similar to the **try-catch** blocks you've seen in CS 61A! If any code in your **try** block throws an exception, it can be "caught" by a **catch** block!

Note: The **catch** blocks would only catch exceptions thrown in the **try** block. If any exceptions are thrown in a **catch** block, the code would terminate with the exception!

```
try {
     //code that produces the exception
     throwException();
} catch (NullPointerException e) {
     //code that handles the specific exception that is thrown
} catch (Exception e) {
     //code that handles any type of exception that is thrown
}
```

The following code below would not compile since all exceptions would be caught by the first **catch** block. The compiler would complain because the second **catch** block would never run! Therefore, a general rule of thumb when creating **try-catch** blocks is to catch specific errors first before addressing more general errors.

```
try {
     throwException();
} catch (Exception e) {
     System.out.println("This catch block would catch all
     exceptions.");
} catch (NullPointerException e) {
     System.out.println("This catch block would never be
     reached.");
}
```

Some further clarification regarding exceptions:
Exception is considered a superclass of all exceptions, unchecked (disregarding **Error** objects) and checked. This means a **catch** block that catches an **Exception** type object could catched checked exceptions.

The class **RuntimeException** extends **Exception**. Any unchecked exception object, disregarding any **Error** object, such as **NullPointerException** extends **RuntimeException**.

Try-Catch-Finally Block (out of scope)

As seen on the worksheet, you can also have a **`finally`** block after your **`try-catch`** blocks. The **`finally`** block must run *before the code terminates*! In the code below, if we threw an exception or had a return statement in our **`catch`** block (which would cause our code to terminate), we would run through the **`finally`** block *before* terminating our code. If the **`finally`** block terminates our code with an exception or return statement, the code will terminate there.

```
try {
     throwException();
} catch (Exception e) {
     System.out.println("Caught exception!");
     throw e;
} finally {
     //runs before the code terminates
     System.out.println("Running finally block!");
     throw new NullPointerException();
}
```

In the terminal, running the code above would yield:

```
    Caught exception!
    Running finally block!
    java.lang.NullPointerException
```

Creating and Throwing Exception Objects

```
try {
     throw new RuntimeException();
     //the code above is equivalent to:
     //RuntimeException e = new RuntimeException();
     //throw e;
} catch (Exception e) {
     throw e;
}
```

Since **`RuntimeException`** extends **`Exception`**, the **`RuntimeException`** would be caught by the **`catch`** block. In the **`catch`** block, the same object with static type **`RuntimeException`** and dynamic type **`RuntimeException`** will be thrown.

Resources

Exceptions Explanation
Advantages of Exceptions
Practice Worksheet on Access Control, Iterators, and Exceptions   Solutions