

Exceptions

Mason Vail

Boise State University Computer Science

Irritatingly Familiar?

```
java.lang.NullPointerException  
    at ExceptionScope.level3(ExceptionScope.java:61)  
    at ExceptionScope.level2(ExceptionScope.java:50)  
    at ExceptionScope.level1(ExceptionScope.java:18)  
    at Propagation.main(Propagation.java:17)
```

or

```
java.lang.ArithmeticException: / by zero  
    at ExceptionScope.level3(ExceptionScope.java:61)  
    at ExceptionScope.level2(ExceptionScope.java:48)  
    at ExceptionScope.level1(ExceptionScope.java:18)  
    at Propagation.main(Propagation.java:17)
```

What Are Exceptions?

Exceptions are objects that represent exceptional situations that prevent the program from continuing in the standard flow of control.

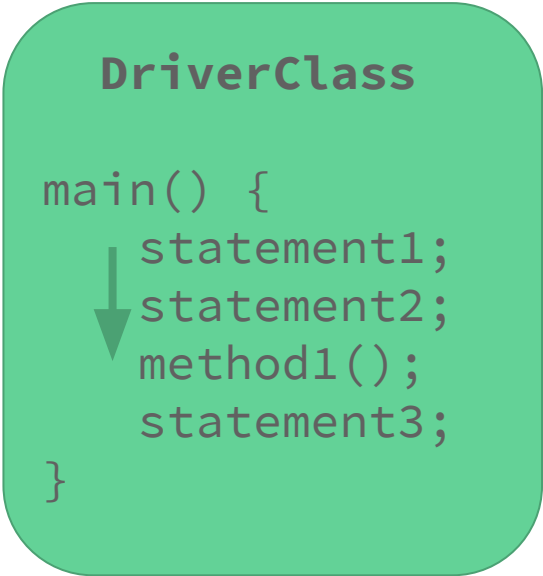
At the root of the Exceptions class hierarchy is the Exception class, itself. From there, descendant classes get more specific about the particular exceptional situation they represent.

Exceptions are descendants of a class called Throwable.

Standard Flow of Control

DriverClass

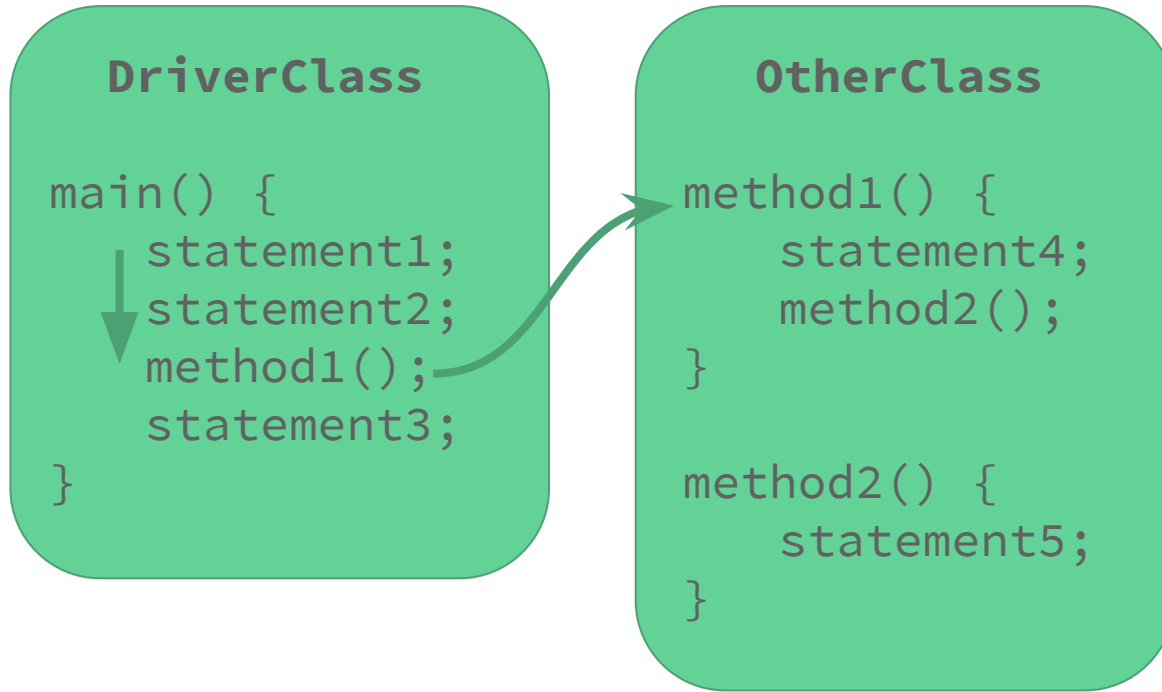
```
main() {  
  statement1;  
  statement2;  
  method1();  
  statement3;  
}
```



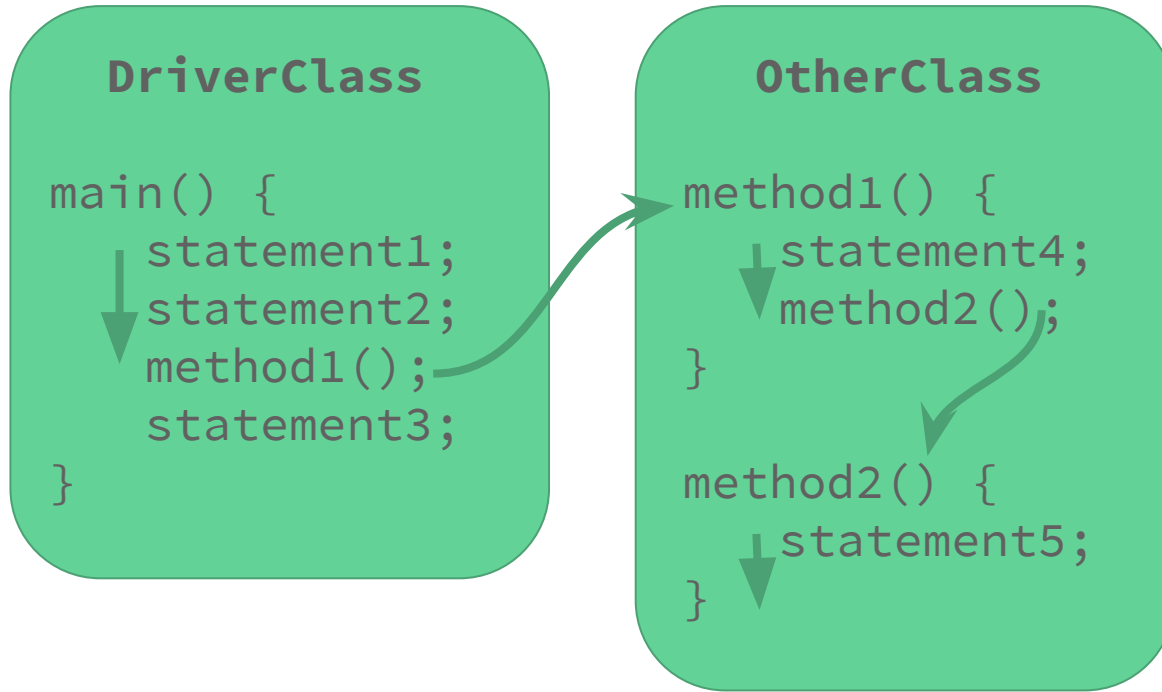
OtherClass

```
method1() {  
  statement4;  
  method2();  
}  
  
method2() {  
  statement5;  
}
```

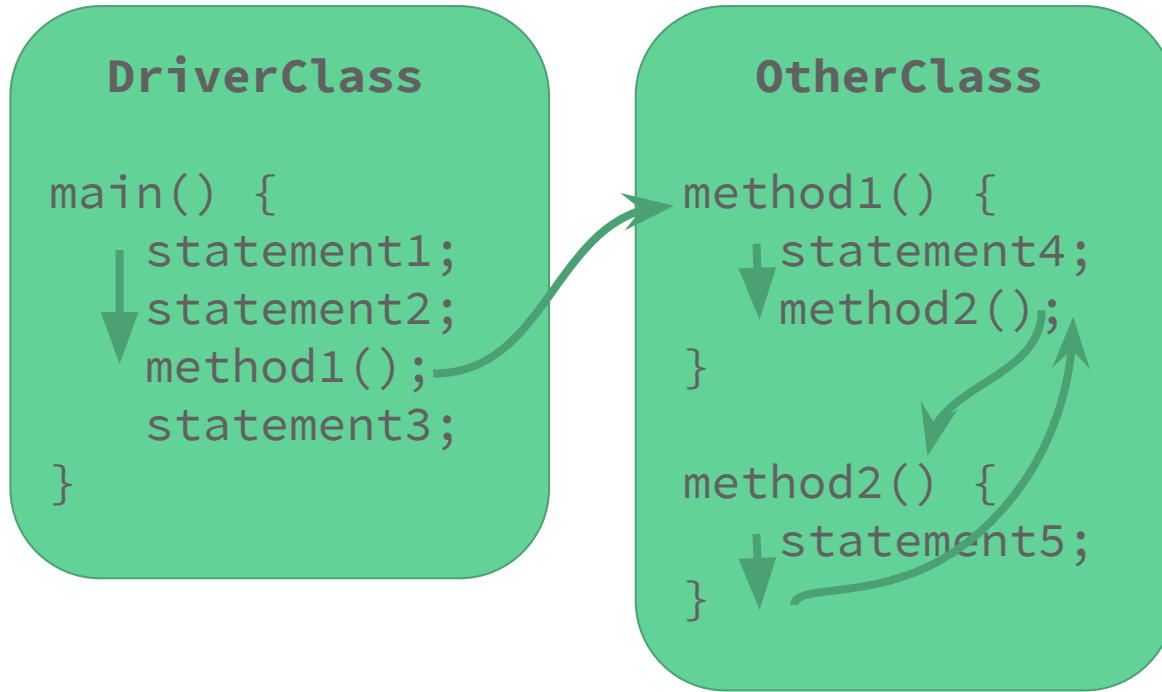
Standard Flow of Control



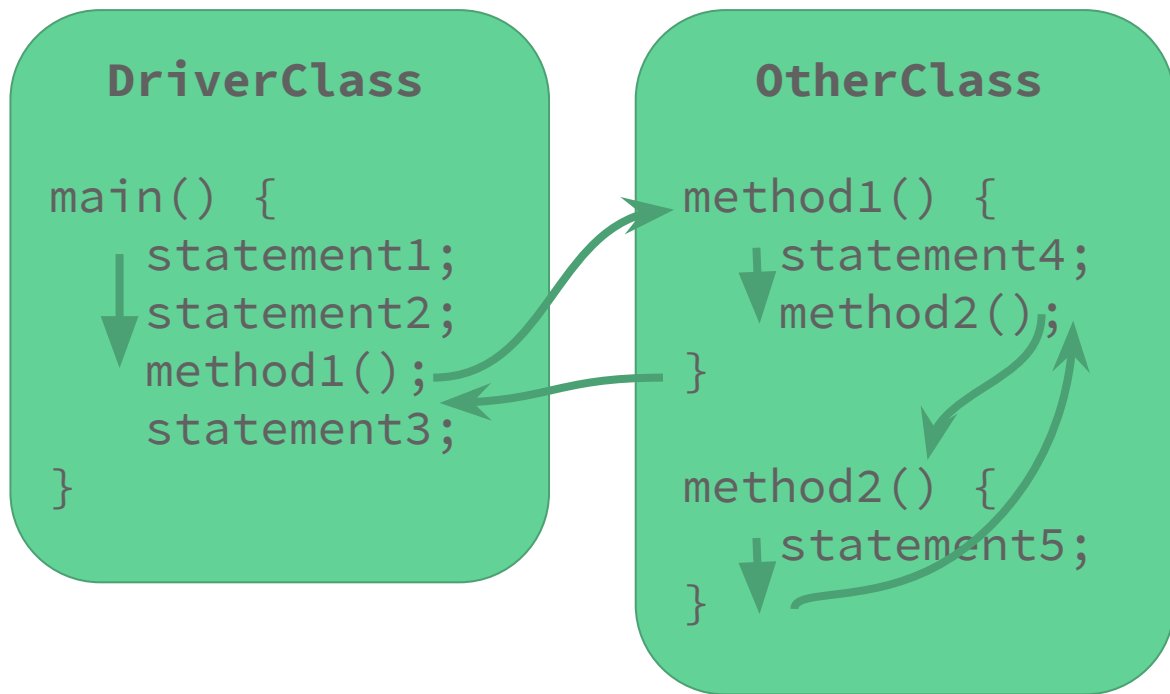
Standard Flow of Control



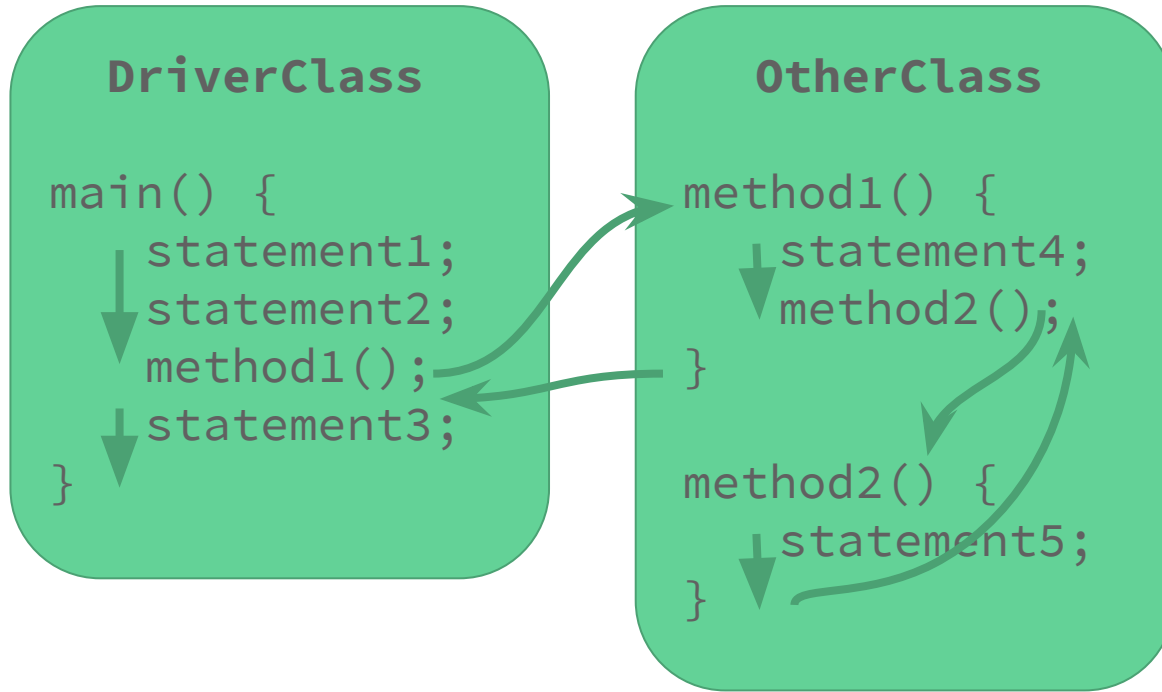
Standard Flow of Control



Standard Flow of Control



Standard Flow of Control



Call Stack

The runtime environment keeps track of the flow of control with a call stack.

When the program begins, the `main()` method is added to the call stack.

When any method is called, it is added to the top of the stack. When a method returns, it is removed from the call stack.

The method currently on top of the call stack is the active method, and all methods deeper in the call stack are waiting for those above to return.

The program ends when `main()` is removed from the call stack.

Standard Flow of Control

DriverClass

```
main() {  
    ↓ statement1;  
    statement2;  
    method1();  
    statement3;  
}
```

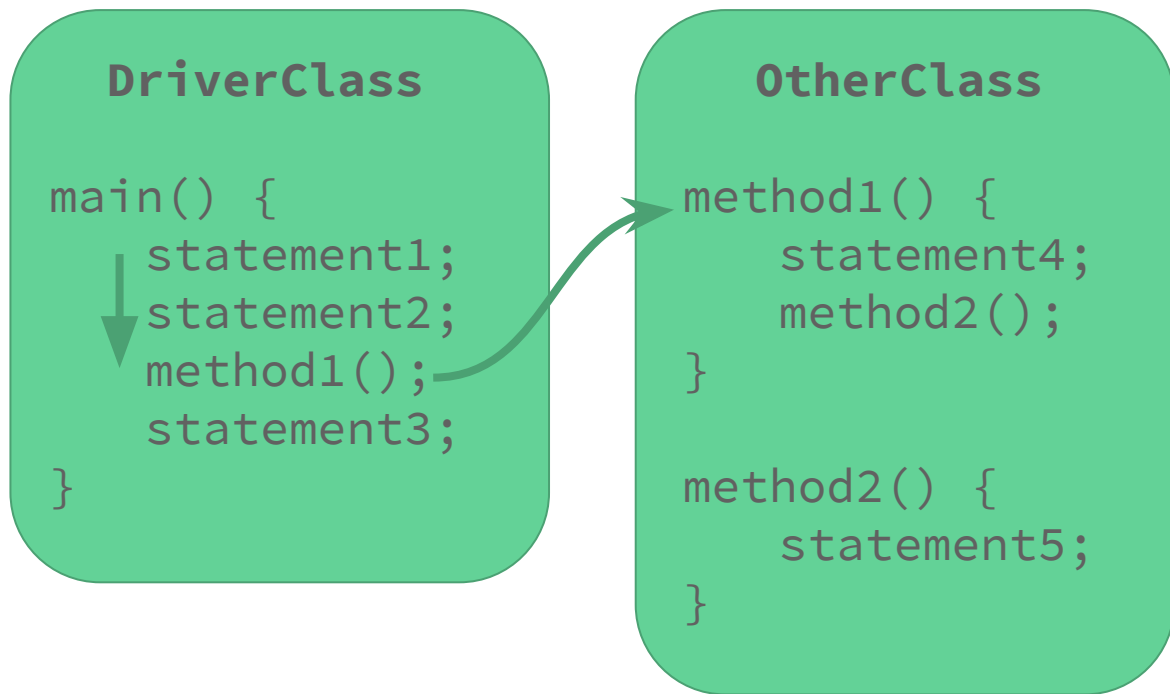
OtherClass

```
method1() {  
    statement4;  
    method2();  
}  
  
method2() {  
    statement5;  
}
```

Call Stack

DriverClass.main()

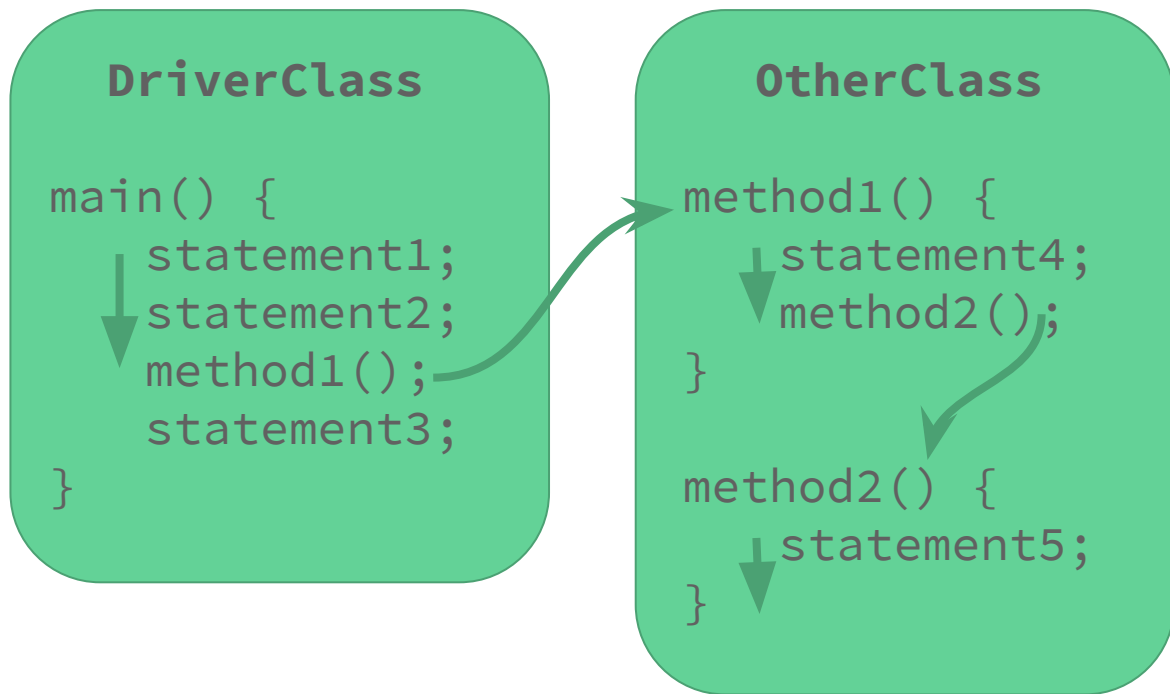
Standard Flow of Control



Call Stack

OtherClass.method1()
DriverClass.main()

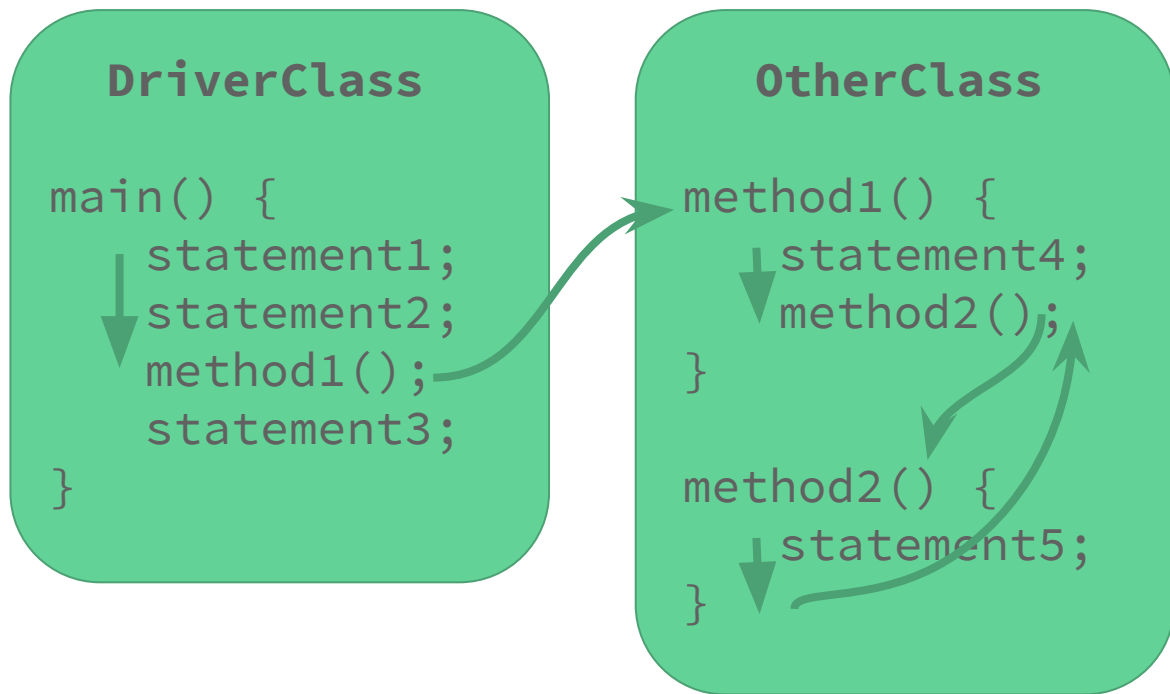
Standard Flow of Control



Call Stack

OtherClass.method2()
OtherClass.method1()
DriverClass.main()

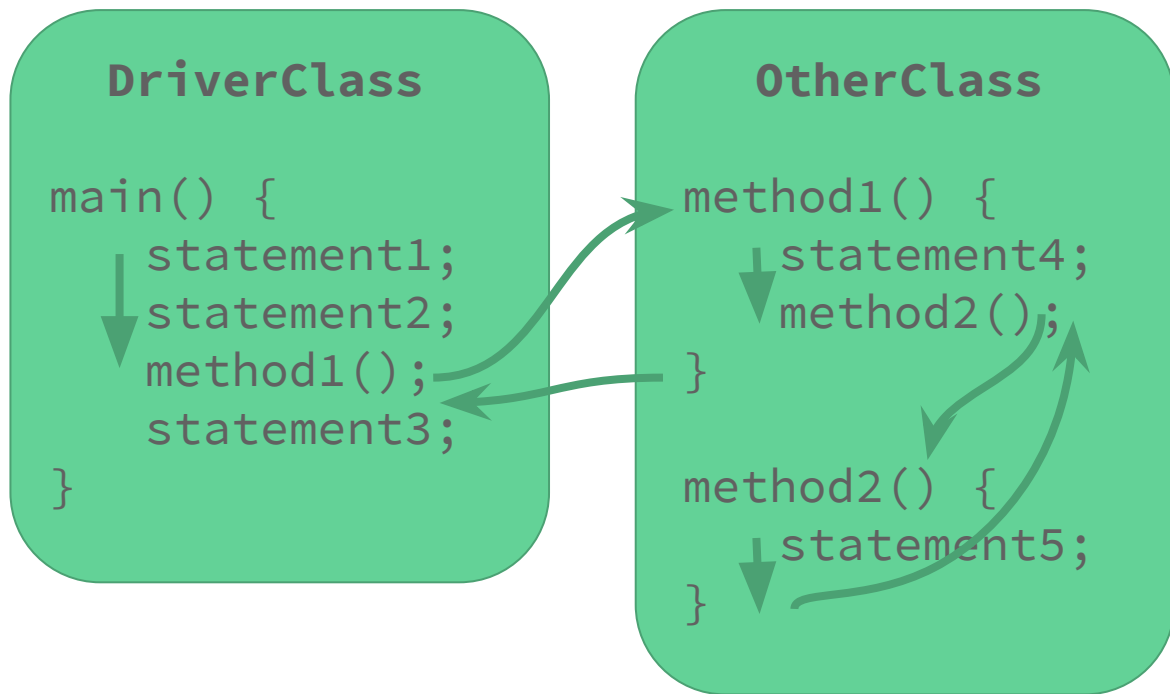
Standard Flow of Control



Call Stack

OtherClass.method1()
DriverClass.main()

Standard Flow of Control



Call Stack

DriverClass.main()

But What If Something Bad Happens?

Sometimes an exceptional situation will be encountered during normal program execution, and the runtime environment cannot continue the standard flow of control.

An appropriate Exception object, representing the program state and the problem, is created and thrown.

The runtime environment enters the exception handling flow of control until the thrown Exception is handled or the program's last chance to handle the Exception has passed and the program crashes.

Exceptions You May Have Seen

NullPointerException

ArithmeticException

IndexOutOfBoundsException

FileNotFoundException

Handling Option 1: Do Nothing

An Exception that is not handled anywhere in the program will result in the program crashing and writing a stack trace to standard output.

Don't ignore that output. It's showing you valuable information about what went wrong and the state of the program when it happened.

Reading a Stack Trace

```
java.lang.ArithmeticException: / by zero
    at ExceptionScope.level3(ExceptionScope.java:61)
    at ExceptionScope.level2(ExceptionScope.java:48)
    at ExceptionScope.level1(ExceptionScope.java:18)
    at Propagation.main(Propagation.java:17)
```

Handling Option 2: Catch the Exception

Handling, or catching, an Exception allows your program to recover and continue.

Dangerous (and dependant) code is placed in a **try** block, followed by one or more **catch** blocks, or handlers.

An optional **finally** block can follow all handlers.

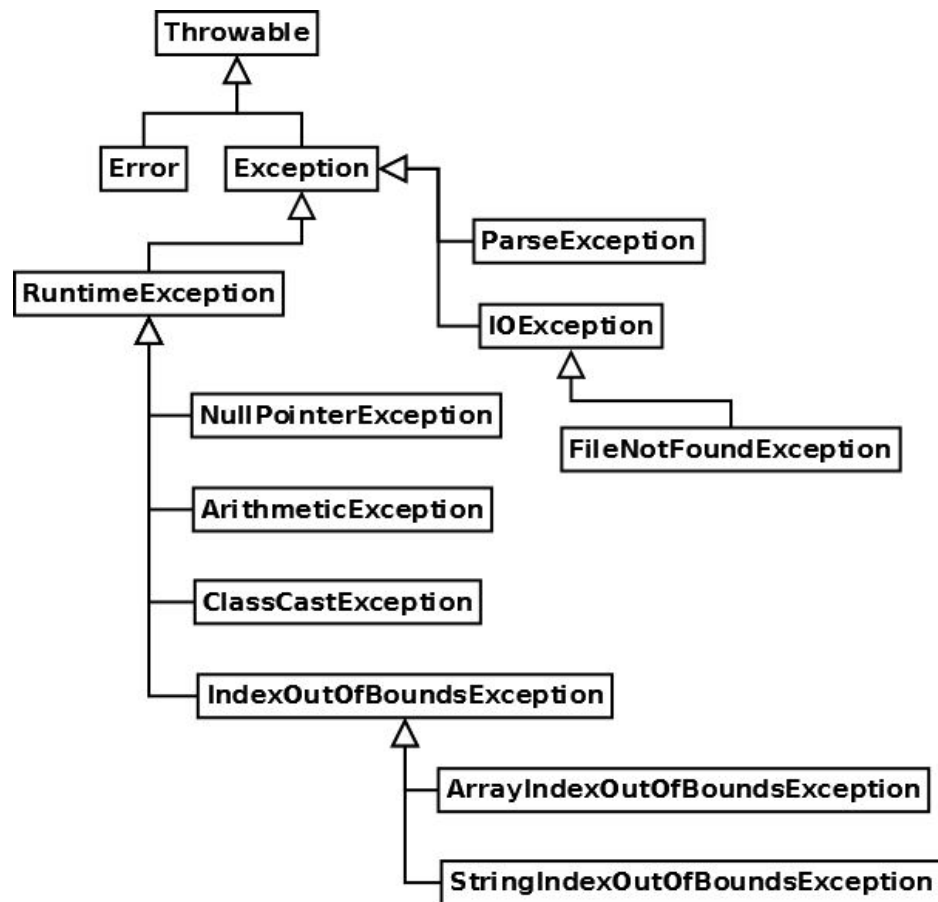
Handling Option 2: Catch the Exception

```
Scanner fileScan = null;
try {
    fileScan = new Scanner ( new File( fileName ) );
    val1 = fileScan.nextInt();
    System.out.println( "Successfully read int " + val1 + " from " +
fileName);
} catch ( FileNotFoundException e ) {
    System.out.println( e.toString() );
} catch ( InputMismatchException e ) {
    System.out.println( fileScan.next() + " isn't an int in " + fileName );
} finally {
    if ( fileScan != null ) fileScan.close();
}
```

Handling Option 2: Catch the Exception

```
Scanner fileScan = null;
try {
    fileScan = new Scanner ( new File( fileName ) );
    val1 = fileScan.nextInt();
    System.out.println( "Successfully read int " + val1 + " from " +
fileName);
} catch ( InputMismatchException e ) {
    System.out.println( fileScan.next() + " isn't an int in " + fileName );
} catch ( Exception e ) {
    System.out.println( e.toString() );
} finally {
    if ( fileScan != null ) fileScan.close();
}
```

Exceptions Hierarchy



Handling Option 3: Propagate the Exception

Rather than handle the Exception itself, a method can propagate the Exception to its predecessor in the call stack. A method that does this is removed from the call stack.

Propagation continues until the Exception is caught or the call stack is emptied.

Checked Exceptions require the **throws** keyword in the method header of each propagating method.

Handling Option 3: Propagate the Exception

```
public static void main(String[] args) {  
    try {  
➡ readVals ( "noSuchFile" );  
    } catch ( FileNotFoundException e ) {  
        System.out.println( e.toString() );  
    }  
}
```

Call Stack

readVals()

main()

```
private void readVals ( String fileName )  
    throws FileNotFoundException {  
➡ Scanner fileScan = new Scanner(new File(fileName));  
    int val1 = fileScan.nextInt();  
    System.out.println( "Read int " + val1);  
    fileScan.close();  
}
```

Handling Option 3: Propagate the Exception

```
public static void main(String[] args) {  
    try {  
➡ readVals ( "noSuchFile" );  
    } catch ( FileNotFoundException e ) {  
        System.out.println( e.toString() );  
    }  
}
```

```
private void readVals ( String fileName )  
    throws FileNotFoundException {  
    Scanner fileScan = new Scanner(new File(fileName));  
    int val1 = fileScan.nextInt();  
    System.out.println( "Read int " + val1);  
    fileScan.close();  
}
```

Call Stack

main()

Writing New Exceptions

You can write your own Exception classes if there isn't one that fits your needs, already.

Extend the most appropriate parent Exception class, give your class a meaningful name, and write a constructor that takes a message and passes it through to the `super()` constructor.

Writing New Exceptions

```
/** Used when a file is found to be in the wrong format
 */
public class InvalidFormatException extends IOException
{

    /** Constructor */
    public InvalidFormatException( String message ) {

        super( message ); // pass through message to super

    }

}
```

Throwing an Exception

You can throw Exceptions, too.

```
throw new Exception( “Just because I can” );
```

```
//Scanner scan expects the next value to be an int
```

```
if ( ! scan.hasNextInt() ) {
```

```
    throw new InvalidFormatException( “missing expected  
    integer” );
```

```
}
```

Exceptions

Mason Vail

Boise State University Computer Science