Object-Oriented Programming

Exceptions

Dr. Seán Russell
sean.russell@ucd.ie

School of Computer Science,
University College Dublin

November 26, 2018

Learning outcomes

After this lecture and the related practical students should...

- understand the difference between syntactic, semantic and resource errors
- understand the concept of the stack and stack traces
- understand the terminology of error recovery
- be able to implement error recovery code for in programs

Table of Contents

Errors

- Writing programs is not simple or easy
- Most, if not all, large programs that are written will contain some errors
- There are many examples of this in the area of game development, titles such as Assassin's Creed: Unity have been know to launch with issues such as characters missing faces, players falling through the map and poor game performance

Types of Errors

There are three main types of errors in programming:

- Syntactic errors
- Semantic errors
- Input/resource errors

Table of Contents

Formal Language

- Programming languages are **formal** languages, this means that there is an exact definition of what is **acceptable** text for a program in a given language
- The actual definition is generally only of interest to language designers and are usually specified in Backus–Naur Form (BNF)
- Almost all programmers simply learn the syntax that is acceptable based on examples and experience with the language, however if interested the syntax can be found in BNF-style at `https://docs.oracle.com/javase/specs/jls/se8/html/jls-19.html`

Syntactic Errors

- Syntactic can be detected automatically
- The most difficult part about syntactic errors is understanding the meaning of the error message
- Language developers attempt to make these easy to understand, however there are so many different types of errors that often the message is complicated
- This issues are generally overcome as the programmer gains experience

Table of Contents

Semantic Errors

- Semantics of a program are about its meaning
- Programs will always do exactly what the source code specifies
- Technically, semantic errors will include mistakes such as incorrect types i.e. trying to multiply an integer by a string
- An example of a semantic error would be a method that adds two numbers when it was supposed to subtract them or code that attempts to access an index outside of an array
- The issue that we are dealing with is that of the correctness of the program

Table of Contents

Input/Resource Errors

- Resource errors can be the most difficult to solve
- A program might function correctly most of the time, but rely on things outside of the control of the programmer
- It might require user input in a specific format or it might need to access a file or access a server
- If a requirement is not met, the program may crash
- The main issue we are dealing with in resource errors is the idea of robustness of the program.

Table of Contents

Errors, Failures and Exceptions

- A semantic coding error might go unnoticed until it causes some problem
- Then the Java interpreter will raise an exception
- An exception is a fatal error that causes the Java interpreter to stop and report the error
- To review the idea of exceptions, we will go through an example program that contains a small bug

Example

```
1  public class ArrayTest {
2    public static int findMaxIndex(int[] a) {
3      int index = -1;
4      int max = 0;
5      for (int i = 0; i < a.length; i++) {
6        if (a[i] > max) {
7          index = i;
8          max = a[i];
9        }
10     }
11     return index;
12   }
```

Example

```
14    public static void main(String[] args) {
15      int[] ex = new int[] { 56, 45, 2, 7, 12, 57, 8 };
16      int mi = findMax(ex);
17      System.out.println("Max is " + ex[mi]);
18    }
19  }
```

Flawed Example

- This code looks correct, and mostly it will work
- However, if we replace the array with
  `ex = new int[] {-56, -45, -2, -7, -12};`,
  something else happens
- Instead we get this output;

```
Exception in thread "main"
    java.lang.ArrayIndexOutOfBoundsException: -1
     at ArrayTest.main(ArrayTest.java:17)
```

Meaning of Output

- This is an example of the output we will see when an exception has happened
- To properly understand the output generated by exceptions we have to have an idea of how the programming language works when any method can call another method
- Every exception is represented by a class and the first line of the output will tell us which exception
- Some exceptions may include some extra information to help solve the problem
- `ArrayIndexOutOfBoundsException` tells us what index the code was trying to access index `-1`

Table of Contents

The Stack

- Whenever we declare a variable in a method, this must be stored somewhere in memory
- However, when we have many methods, there will often be many variables with the same name
- The problem is solved by using a stack to store these variables
- When a method is executed, it is given a position on the stack to store information

The Stack

- When the main method is executed it is given this space and stores the variables `ex` and `mi` there
- Whenever another method is called, a new position is created and placed on the stack
- The variables in all other positions can not be accesses until this method is finished
- In our example, the method `findMax` gets a position that is placed on top of the stack, and it stores the variables `index`, `max`, `i` and a reference to `ex` with the name `a`
- While the code is executing it can access and change any of these variables, but no others

The Stack

- When this method is complete and has returned its result, it no longer needs these variables and the position that was created for the method is removed
- This means that the next position is now moved to the top of the stack and the main method has access to its own variables
- This process can go for many level deep, where one method calls another and that calls another
- This information is presented to us whenever an exception occurs
- The output from our previous example is called a stack trace

Stack Trace

- When an exception happens, a stack trace gives us the state of the stack at the exact moment that the exception occurred
- This gives us details like the class and method name as well as the line number for each level of the stack
- The Items in the stack are printed in the order that they are removed
- This means that the bottom of every exception will always be the `main` method
- The line number tells us exactly what code was being executed at the time
- This allows us to identify where the problem was

Example Changed

```
1  public class ArrayTest {
2    public static int findMax(int[] a) {
3      int index = -1;
4      int max = 0;
5      for (int i = 0; i < a.length; i++) {
6        if (a[i] > max) {
7          index = i;
8          max = a[i];
9        }
10     }
11     return a[index];
12   }
13
14   public static void main(String[] args) {
15     int[] ex = new int[] {-56, -45, -2, -7, -12};
16     int max = findMax(ex);
17     System.out.println("Max is " + max);
18   }
}
```

Example Changed

- If we execute this program we get the following output:

```
1 Exception in thread "main"
    java.lang.ArrayIndexOutOfBoundsException: -1
2     at ArrayTest.findMax(ArrayTest.java:11)
3     at ArrayTest.main(ArrayTest.java:16)
```

- This tells us that we tried to access the array at index -1 while we were executing line 11 in the method `findMax`, which was called in the `main` method on line 16
- This allows us to see exactly where the problem is in our code and fix it

Table of Contents

Error Recovery

- Ideally, our programs should recover from exceptions
- Before we do this we should look at some of the terminology used to describe error handling
- When Java has a problem with our code, it throws an exception
- An exception that is thrown can be caught at any point in the stack
- Catching an exception is how we can recover from exceptions that happen in our program

try-catch

- In Java we use a complex statement called a `try-catch` to recover from errors
- There are two sections in a try-catch statement;
  - The try section, where we attempt the code the might cause a problem by throwing an exception
  - The catch section, where we describe what should be done when that exception happens

```
1 try{
2   // attempt the code
3 } catch(Exception e){
4   // what to do when there is a problem
5 }
```

Solution to Problem

- A try-catch statement could be used to solve the problem in our example, where we try and access the array and if something goes wrong return the value 0

```
1 try{
2    return a[index];
3 } catch(Exception e){
4    return 0;
5 }
```

- The above code means that any exception caused by this code is dealt with immediately and the program will not crash
- However, this is not a good solution

An Appropriate Solution

- The problem is caused by a semantic error and rather than catching the error when it happens, we should instead fix the error
- The code is trying to access index -1
- This means that the value of index is not set at any point in the loop
- By looking at the numbers in the array we can see that they are all negative and the value of max starts at 0
- This means that none of the values are bigger than max and the max value is never set
- Instead of 0 we should set the value of max to be `Integer.MIN_VALUE`

Table of Contents

Robust Programs

- A program is robust if it recovers from unexpected errors
- By 'unexpected errors' we mean resource errors beyond the control of the programmer
- Generally, a robust program will catch any exceptions when things go wrong and
  - Tell the user if necessary
  - Continue executing if possible

Robust Program Example

- For example, assume we are writing a program to read student data from a file
- If the user enters the name the file incorrectly, the the program will not be able to find the data
- Without error recovery this would cause the program to crash
- If instead we recover from the error by allowing the user to enter the file name again we can continue executing
- This is an example of robust behaviour

Table of Contents

Exception Objects

- In every catch statement there is a variable that contains information about the exception that was thrown
- When we define the code `catch (Exception e)`, we can use the object `e` to find or output information about the exception that happened
- When an exception is not caught, the stack trace for that exception is printed and execution is stopped
- However, when we catch the exception we can still display the stack trace and continue execution

Printing a Stack Trace

- If we have an exception named `e`, then we can call the method `printStackTrace` to have the stack trace of the exception output to the screen e.g. `e.printStackTrace()`
- This will give us all of the information about the program when the exception was thrown
- This is particularly useful for debugging the error recovery code we write
- However, generally this code should be replaced by some sort of error recovery mechanism when writing code commercially.

Table of Contents

Exceptions

- There are many different types of exceptions, each defined as a class
- The root of the hierarchy is a class called `Throwable`
- However, generally we only deal with two of its know direct subclasses
  - Error
  - Exception

The Error Class

- Errors are caused by serious problems that most applications should not catch
- We will not study these further

The Exception Class

- Exceptions are problems that most programs might want to recover from
- There are two types of classes that extend Exception
  - Checked Exceptions
  - Unchecked Exceptions

Table of Contents

Unchecked Exceptions

- An unchecked exception represents an error that could possibly happen but is generally very unlikely
- An exception is unchecked if it is a subclass (directly or indirectly) of the class `RunTimeException`
- Typical examples of unchecked exceptions are `IndexOutOfBoundsException`, `NullPointerException` or `InputMismatchException`
- Programmers are not required to implement error-recovery code when these exceptions are possible

Why no Requirement

- If we think about the three examples, it is easy to see why we are not required to implement error-recovery when these exceptions are possible
  - An `IndexOutOfBoundsException` is possible every time we use an array. Imagine how complicated our code would be if every time we wanted to access an array we needed to use a try-catch statement.
  - A `NullPointerException` is possible every time we use an object (accessing an instance variable or calling a method).
  - An `InputMismatchException` is possible any time we try to read information of a type from a String (or the user or a file).

Table of Contents

Checked Exceptions

- A checked exception represents an error that is likely to happen when the code is executed
- An exception is checked if it is a subclass of `Exception`, but not a subclass of `RunTimeException`
- Programmers are required to implement error-recovery code whenever a checked exception is possible

Checked Exceptions

- Typically, this means code that is performing an action, such as file or network access, that is likely to have a problem
- If we want to read a file, there are many problems that can happen such as its location has changed or the format is incorrect
- If we want to access information on an internet server, there are many problems that can happen such as a problem with our internet connection or a problem with the server connection
- In most applications, error recovery code is only written for checked exceptions

Table of Contents

Recovering from Errors

- The method `parseInt` in the `Integer` class can be used to convert a string containing a number into an integer value
- However, when this method is used it may throw a `NumberFormatException`
- This exception is an unchecked exception, so we are not required to implement error recovery, but we can implement it if we want to

```
1 public static void main(String[] args) {
2   String s = ""; // string with some value
3   try {
4     int x = Integer.parseInt(s);
5   } catch (Exception e) {
6     System.out.println("Not a valid int");
```

Recovering from Errors

- We put the code into the try section, and if there is a problem our program can continue executing
- If the string entered does not contain an integer, then the message will be printed out
- However, the code in the try statement stops at the point where the exception happens
- Any code following that statement will not be executed
- When there is no exception, the code executes normally and the code in the catch section is not executed

Catch Specific Exceptions

- This means that whenever we see a try catch statement, the code can either be executed correctly or an exception will be thrown and the catch statement will be executed
- The code in the example will catch any exception that is a subclass of the `Exception` class because this is the type of exception that we have declared
- However, if we wanted to be more specific we could replace this with `catch (NumberFormatException e)`
- This makes it clearer what exception is expected to happen and be recovered from

Table of Contents

Multiple Catch Sections

- Different types of errors can happen to the same code
- In our example `Integer.parseInt(s)`, there are two possible exceptions, the `NumberFormatException` if the string does not contain a number and a `NullPointerException` if the variable s is null
- It is likely that we will want to recover from these errors in different ways, in this case we can add multiple catch sections to the same `try-catch` statement
- This allows us to specify what we want to happen for each type of error

Catching Multiple Exceptions

```
1  public static void main(String[] args) {
2     String s = null; // string with some value
3     try {
4        int x = Integer.parseInt(s);
5     } catch (NumberFormatException e){
6        System.err.println("Not a valid int");
7     } catch (NullPointerException npe){
8        System .err.println("s has no value");
9     }
10 }
```

Order of Catch Statements

- When an exception is thrown in a try section, the JVM will look through the list of exceptions that are caught in order until it finds one that matches
- This includes any superclass Exceptions that may be declared
- For example, if the first catch statement is for `Exception` and the second is for `NumberFormatException`, then the first will always match because `NumberFormatException` is a subclass of `Exception`

Finally

- Optionally, after the catch sections we can also include a `finally` section
- This section will be executed both if an exception is thrown and if it is not
- These are usually used to manage resources that are used in the try section of the statement
- For example, if we are using a network stream, this should be closed after use
- Placing this code in a `finally` section allows the code to be executed and the stream closed both if there is a problem and if there is not

Table of Contents

Checked Exception

- Checked exceptions are exceptions that are not subclasses of `RunTimeException`
- A programmer is required to implement error-recovery when these exceptions are possible
- For example, the `IOException` class 'signals that an I/O exception of some sort has happened'
- This class has even more specific subclasses that include
  - FileNotFoundException
  - MalformedURLException
  - RemoteException

Checked Exceptions

- Whenever it is possible that a checked exception might be thrown by code in a method, the programmer must either
  - catch the exception using a try-catch statement (called handling) or
  - pass the exception to the class that called it
- Passing the exception on is called advertising
- This now means that the other class will now face the same choice.

Example

- Lets consider the method `read` in the class `BufferedInputStream`
- The declaration of this method is

  ```
  public int read() throws IOException
  ```

- This means that the method may throw an `IOException`
- The following code tries to use the read method, but it does not catch the error or advertise it, because of this the code will not compile

```
1 public int readInt(BufferedInputStream b) {
2   return b.read();
3 }
```

Example Handled

- If we change the so that the exception is handled, then the code will compile

```
1 public int readInt(BufferedInputStream b) {
2   try {
3     return b.read();
4   } catch (IOException e){
5     return -1;
6   }
7 }
```

Example Advertised

- If instead we decided that, we wanted to advertise the exception and not put any error handling code we could advertise it instead
- This is done by adding `throws` and the name of the exception to the end of the method definition

```
1 public int readInt(BufferedInputStream b) throws
    IOException {
2   return b.read();
3 }
```

- This might seem like it is the easiest solution
- Because the `readInt` may throw an `IOException` any code that uses it must have error-recovery code

Catch or Advertise

- Choosing whether to catch or advertise is a design decision
- If we want our programs to be robust, there must be error-recovery code somewhere
- The best practice is to identify the most appropriate place to handle the exception
- Wherever we choose, there should be enough information to write a meaningful error message and make a reasonable decision as to if and how to **continue** execution

Catch or Advertise

- In the example above it is impossible to know which is the best choice without knowing what the code will be used for
- If we assume that the code was being used to read the grades of students from a file, then returning -1 when a grade cannot be read would be bad
- If the class calling the method knew that -1 meant a missing grade and checked, it would be OK
- If the exception is advertised, then the entire file reading operation might be interrupted by an exception instead of just reading a single number

How to Continue Execution

- The whole idea behind error recovery is that when an error happens that we are able to recover from it
- This is not always possible
- When we do have a problem, what do we do?
- Lets take an example of a method that is supposed to perform some operation and the return the result
- If some error happens while the method is executing we have two options, we can return a special value like -1 or null, or we can throw an exception and allow problem to be solved somewhere else

Returning a Special Value

- This is a very common solution to the problem
- However, it is very important that this is explained in the documentation of the method
- This way any programmer that uses your method will be aware that if something goes wrong the special value is returned and they can check for it

Table of Contents

Throwing an Exception

- When we encounter a problem and want to throw an exception, we have some choices to make
- First, we must decide what type of exception to throw
- Second we must decide where the exception should be thrown and lastly we may need to decide where the exception should be caught
- When facing a potential problem in our code, it is often easier to create our own exception to describe the problem
- An exception is a class just like any other, we simply need to extend `Exception` and optionally add a small bit of code to explain the exception

Example

- Lets assume that we have been given the task to design and implement a Stack
- While we were writing the pop method, we realised that there could be a problem if the user tried to take an item from an empty stack
- Depending on the implementation, this could cause a `NullPointerException` or an `ArrayIndexOutOfBoundsException`
- Instead we can create an exception to throw

```
1 public class StackEmptyException extends Exception {
2   public StackEmptyException(){
3     super("The Stack is Empty");
4   }
5 }
```

Advertising the Exception

- We can use this new exception in the code of our stack interface to define that this exception might be thrown if the stack is empty and then throw it in the implementation when the problem happens

```
1 public interface Stack{
2   public int top() throws StackEmptyException;
3   public int pop() throws StackEmptyException;
4   public int push();
5   public int size();
6   public boolean isEmpty();
7 }
```

Throwing the Exception

- Now we have declared that the methods `pop` and `top` both might throw the `StackEmptyException`
- Next we need to add the code to the methods that will actually perform this action
- As an exception is a class like any other, we must construct an object if we want to throw it
- Therefore we need to use the keyword `new` and pass any parameters to the constructor that are required
- This gives us `new StackEmptyException();` to create the object
- Once the object is created, we use the keyword `throw` to throw the exception, however this is only done when we are sure that the stack is empty

Throwing the Exception

```
1 public int pop() throws StackEmptyException{
2   if(isEmpty()){
3     throw new StackEmptyException();
4   }
5   int t = values[top - 1];
6   top--;
7   return t;
8 }
```

Table of Contents

Variable Scope

- We are used to the idea that whenever we declare a variable, it is only accessible from certain places
- For example an instance variable can be accessed by anyone using an object of that type, a local variable can only be accessed inside that method
- A variable declared inside a section of a try-catch statement, is only accessible inside that section
- This means that in our example "Catching an exception in Integer.parseInt" the variable x can only be used inside the try section of the code

Catching an exception in Integer.parseInt

```
1 public static void main(String[] args) {
2   String s = ""; // string with some value
3   try {
4     int x = Integer.parseInt(s);
5   } catch (Exception e) {
6     System.out.println("Not a valid int");
7   }
8 }
```

Variable Scope

- Often this is an acceptable way of declaring and using variable, but sometimes we will want the variable to be accessible after the try section
- To do this, the variable must be **declared** before the try section
- This allows the variable to be used later in our code

```
1 int x;
2 try {
3   x = Integer.parseInt(s);
4 } catch (NumberFormatException e){
5   System.out.println("Not a valid int");
6 }
7 System.out.println("Integer value is "+x);
```