UCL

# COMP1008 Exceptions





UCL

## Runtime Error

- Unexpected error that terminates a program.
  - Undesirable…
- Not detectable by compiler.
- Caused by:
  - Errors in the program logic.
  - Unexpected failure of services
    - E.g., file server goes down.






UCL

## Exception example

```
int x = 1 ;
int y = 0 ;
int z = x / y ;
java.lang.ArithmeticException: / by zero
        at T9.main(T9.java:7)
```

- This message is displayed when the code is executed.
- You typically see a call stack trace.






UCL

## Call Stack Trace

```
apple.awt.EventQueueExceptionHandler Caught Throwable : java.lang.ArithmeticException: / by zero
java.lang.ArithmeticException: / by zero
        at uk.ac.ucl.cs.editor.MemoListEditor.backward(MemoListEditor.java:60)
        at uk.ac.ucl.cs.view.SwingMemoListView$4.actionPerformed(SwingMemoListView.java:173)
        at javax.swing.AbstractButton.fireActionPerformed(AbstractButton.java:1819)
        at javax.swing.AbstractButton$ForwardActionEvents.actionPerformed(AbstractButton.java:1872)
        at javax.swing.DefaultButtonModel.fireActionPerformed(DefaultButtonModel.java:420)
        at javax.swing.DefaultButtonModel.setPressed(DefaultButtonModel.java:258)
        at javax.swing.plaf.basic.BasicButtonListener.mouseReleased(BasicButtonListener.java:247)
        at java.awt.Component.processMouseEvent(Component.java:5166)
        at java.awt.Component.processEvent(Component.java:4963)
        at java.awt.Container.processEvent(Container.java:1613)
        at java.awt.Component.dispatchEventImpl(Component.java:3681)
        at java.awt.Container.dispatchEventImpl(Container.java:1671)
        at java.awt.Component.dispatchEvent(Component.java:3543)
        at java.awt.EventDispatchThread.run(EventDispatchThread.java:100)
        etc.
```






UCL

## `Null` pointer exception

- Should be familiar...

```
java.lang.NullPointerException
        at T10.main(T10.java:7)
```

(So references are really pointers according to the JVM!)

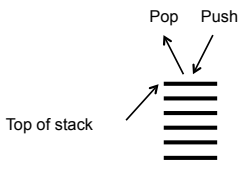





UCL

## Stack

- A stack is a last-in first-out queue:
- Items are pushed on,
- and popped off the pile.
- Can use an ArrayList to store stack contents.



## Class Stack

```
class Stack<T>
{
   private ArrayList<T> contents;
   public Stack()
    { contents = new ArrayList<T>();}
    // Methods push, pop, top
    ...
}
```

This is a generic Stack class. T is a type variable that is instantiated with a real type in a declaration.

```
Stack<Integer> = new Stack<Integer>();
```





## Programmers make mistakes!

```
Stack<Double> s = new Stack<Double>() ;  // Empty stack
Double d = s.pop() ;                     // Whoops!!
System.out.println(s.top()) ;    // What is printed?
```

How can you force someone to take notice when these kinds of errors are made?

Best answer: Use test-first programming.





## (Non) Solution 1

Ignore the problem and hope it never happens!

Relies on object always being used correctly...





## Solution 2

Print out an error message.

```
public Object pop()
{
  if (contents.size() == 0) // empty
  {
     System.out.println("Stack empty");
     return null;
  }
  else
    return contents.remove(0);
}
```

Message *may* be noticed – but which call of pop caused the problem?

What value is returned? Caller must deal with null being returned.





## Solution 3

Stop the program!

```
public T pop()
{
  if (contents.size() == 0) // empty
  {
    System.out.println("Stack is empty") ;
    System.exit(1) ;  // Force program to stop
    return null;  // Needed to compile but not used
  }
  else
    return contents.remove(0);
}
```

Bad design to put output into data structure class like this. Don't do it!

- Can't avoid noticing this!
- But still little information about where and why.





## Solution 4

```
public T pop()
{
  if (contents.size() == 0)  // empty
  {
    return null; // "silent" default action if empty
  }
  else
  {
    return contents.remove(0);
  }
}
```

- Take some default action and rely on client code doing right thing with result.
- But moves problem somewhere else, *without* compiler checking correctness.

## Solution 5

Throw an exception!

```
public T pop() throws EmptyStackException
{
    if (contents.size() == 0) // empty
    { throw new EmptyStackException(); } // Note no return needed here
    else
    { return contents.remove(0); }
}
```

- Force the program to deal with the error or terminate.
- Force the compiler to check code.
- Force the programmer to write the code properly.





## Java Exception Mechanism

- To allow program execution to continue after an error.
- Or, at least, terminate gracefully.

- Uses five keywords in the language:
  try, catch, throw, throws, finally





## Try and catch

```
try
{
  a.doSomething() ;
}
catch (Exception e)
{
  // Handle the exception
}
```

A *try block* tries to execute statements.

A *catch block* catches exceptions thrown from the try block.





## Throw

```
throw
new MyException (“Method doSomething failed”) ;
```

- The throw statement throws an exception.
- It takes an *exception object reference* as an argument.
- Somewhere a catch block must catch the exception.





## Catch

```
catch (Exception e) { ... }
```

- This catches an object of library class Exception or *any* of its subclasses.
- Typically, your exceptions are *subclasses* of class Exception.

```
catch (MyException e) { ... }
```

A subclass extends another class. Exception represents exceptions in general, a subclass represents a specific kind of exception like EmptyStackException.





## Multiple catch expressions

- Several catch expressions can be given to catch a range of exception objects of different classes:

```
catch (MyException e1) { … }
catch (NumberFormatException n) { … }
catch (InvalidDataException i) { … }
```

## Finally `finally`

```
try
{
  f() ;
}
catch (MyException e) // optional
{
  // Do something
}
finally
{
  // Guaranteed to execute
  // this whatever happens.
}
```

A finally block will be always be executed regardless of what else happens.






## Standard Exception Classes

- Throwable
  - Superclass of all exception classes.
- Error (extends Throwable)
  - Serious error that is not usually recoverable.
- Exception (extends Throwable)
  - Error that must be caught and recovered from.
- RuntimeException (extends Exception)
  - Error that may be caught if desired.






## class Throwable

- Throwable provides:
  - A String to store a message about an exception.
  - A method String getMessage() to return the message string.
  - A method printStackTrace.
  - And a few other methods.
- Subclasses extend Throwable and can add further variables and methods.
- Most, but not all, Exception classes represent exceptions that *must* be caught.
  - The compiler will check.
  - A small number of Exception classes represent exceptions that do not need to be caught.






## Writing an exception class

```
class MyException extends Exception
{
  public MyException ()
  { super("Default message") ; }

  public MyException (String s)
  { super (s) ; }
}
```

The extends keyword specifies that MyException is a *subclass* of Exception.

We will be looking at subclasses in detail later in the course.






## Questions?






## An Example

```
public void f(String s)   // s should represent an integer.
{
    int tmp ;
    try
    {
       tmp = Integer.parseInt(s) ; // This can fail
    }
   catch (NumberFormatException e)
   {
       tmp = -1 ; // Set tmp to some default value
   }
   // Carry on and use tmp
}
```

## Integer.parseInt

```
public static int parseInt (String s)
    throws NumberFormatException
{
  return parseInt(s,10) ;
}
```

- parseInt is overloaded and calls another version of parseInt that can throw an exception.
- This version does not catch the exception but declares that it can occur.






## Throws keyword

- A method can directly or indirectly throw an exception without catching it.
  - the exception must be declared by a throws declaration,
  - for class Exception and subclasses (excluding RuntimeException).
- If not, the compiler will fail the code.
  - Guarantees that exception will be used correctly.






## Integer.parseInt (2)

```
public static int parseInt(String s, int radix)
      throws NumberFormatException
{
  if (s == null)
    {
      throw new NumberFormatException("null");
    }
  if (radix < Character.MIN_RADIX)
  {
      throw new NumberFormatException
      ("radix " + radix + " less than
       Character.MIN_RADIX");
  }
```






## Integer.parseInt (3)

- For every invalid state detected in the method body an exception is thrown.
- The method does not attempt to catch its own exceptions.
- Catching is left to the caller of the method.
  - Or caller of the caller of the method





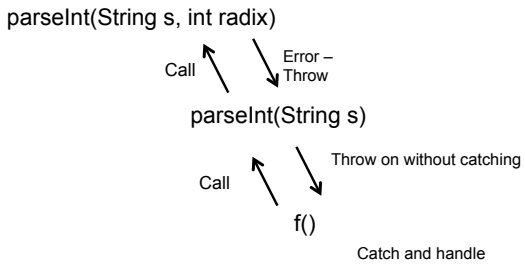

## Integer.parseInt (4)

- When an exception is thrown the method terminates immediately.
  - without returning a value in the normal way.
- The exception is thrown back to the calling method.
- The caller method either:
  - catches the exception.
  - or it terminates immediately and throws the exception back to its caller.






## Integer.parseInt (5)



## Propagating exceptions

- Passing an exception up through a *chain of active method calls* is called *propagation*.
- The active methods calls are those still in progress, leading to the point where the exception occurred.






## Uncaught Exceptions

- If an exception is passed back to the main method without being caught the program will terminate with an error.
  - However, program won't compile if exception must be caught.
    - I.e., subclass of Exception (excl. RuntimeException).
- Good design and testing practice will avoid this happening...






## Stack class revisited…

- Define a StackException class.
- Any Stack method that can fail should throw an exception (e.g., empty/full stack).
- Stack does *not* catch its own exceptions.
- Clients of Stack *must* be prepared to catch the exceptions.
  - The calling method, or a method that calls it, must have a catch block for the exception.






## Stack client

```
public void aMethod(Stack<String> aStack)
{
  String s;
  try
  {
    s = aStack.pop();
  }
  catch (StackException e)
  {
    // Do something to recover
    // from problem
  }
  // ... rest of method
}
```

or

```
public void aMethod(Stack<String> aStack) throws StackException
{
  String s = aStack.pop();
  // ... rest of method
}
```

The method or a method that calls it must contain the catch block.






## When to use Exceptions

- The normal sequence of events fails.
  - I/O and user action.
  - Method cannot proceed and there is no practical return value (e.g., parseInt).
  - Need to return control to a method at top of call stack.
- *Not* a substitute for using return.






## Issues…

- Too many exceptions require too many try/catch blocks.
  - Complicates code.
  - Can reduce readability.
  - Many methods need throws declaration.
- But can simplify.
  - Some code is written assuming no errors, so simpler.
- A balance is needed.

## Local v. Global

- Some local operation (parseInt, open file) may fail.
  - Deal with problem locally and proceed.
- Top level method(s) catch exceptions from anywhere in program.
  - Terminate current operation but leave program running.
  - Top level strategy for handling errors (e.g., save data).






## More Information

- Read the text book.






## Summary

- Exceptions allow errors to be represented and handled in a safe way.
- Java uses the try, catch & throw mechanism.
- Throwing an exception forces client code to do something.
- Don't forget finally.