Programming 2: Lecture 3.

Nuts and Bolts

Serialisation pages 1-5

Exceptions
Pages 7-12

Reflection
Pages 5-7

```
                          Windows

An exception  06 has occured at 0028:C11B3ADC in VxD DiskTSD(03) +
00001660.  This was called from 0028:C11B40C8 in VxD voltrack(04) +
00000000.  It may be possible to continue normally.

*  Press any key to attempt to continue.
*  Press CTRL+ALT+RESET to restart your computer.  You will
   lose any unsaved information in all applications.

                 Press any key to continue
```

Serialisation

Saving the state of an object to file so it can be easily reloaded

Serialisation

- Serialisation provides the ability to save the state of an object beyond the life of the program and the virtual machine
- Objects are “flattened” into a bytecode file so they can be easily loaded later
- Java provides a Serialization API that makes it all very easy
- **The Class of the object to be serialised must implement the `Serializable` interface**
- This interface is completely empty! Its just a marker interface

Serialization Interface

```
import java.io.Serializable;
import java.util.Date;
import java.util.Calendar;
public class PersistentTime implements Serializable
{
      private Date time;

      public PersistentTime(){
            time = Calendar.getInstance().getTime();
      }
      public Date getTime(){
            return time;
      }
}
```

Saving to Byte Code

Obects are saved to an `ObjectOutputStream` with the method `writeObject`

Serialised objects have the extension .ser in java

```
import java.io.*;
public static void main(String [] args){
      String filename = "time.ser";
      PersistentTime time = new PersistentTime();
     try{
     FileOutputStream fos =
                        new FileOutputStream(filename);
     ObjectOutputStream out = new ObjectOutputStream(fos);
     out.writeObject(time);
     out.close();
    }
   catch(Exception ex) {
      ex.printStackTrace();
    }
}
```

Loading an Object From Byte Code

Objects are loaded from an `ObjectInputStream` with the method `readObject`

```
try{

   FileInputStream fis = new FileInputStream(filename);
   ObjectInputStream in = new ObjectInputStream(fis);
   time =(PersistentTime)in.readObject();
   in.close();
   System.out.println(" Current time = “
                  +Calendar.getInstance().getTime());
   System.out.println(" Persistent time = "+time);
  }
```

```
Current time = Mon Nov 01 16:36:17 GMT 2013
Persistent time = Mon Nov 01 16:36:00 GMT 2013
```

Default Serialisation

• It is possible to persist Java objects through JDBC and store them into a database or to persist across a network.

•It is **not** possible to persist static fields with serialisation.

•The **`Object`** class does not implement **`Serializable`**.

•If we make an object **`Serializable`**, by default all data fields are saved

•When a serialised Object is read in, the constructor is not called.

• If we change the methods or fields after saving then try reload, we get an **InvalidClassException** exception

•Repeated writes to the same output do not overwrite previous writes. You must close and reopen to do this

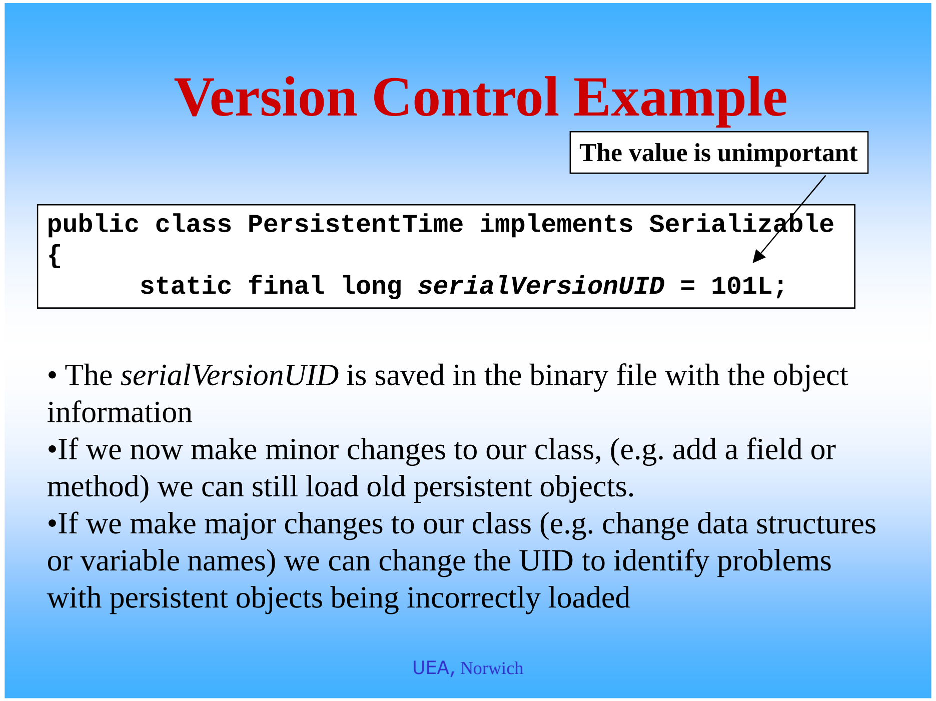
Version Control

•If we change the methods or fields after saving, then try reload, we get an **InvalidClassException** exception

•All serialised classes contain a `serialVersionUID`. This is used by `readObject` to check that it is ok to load (otherwise the default serialVersionUID based on a sum of hashCodes is used)

•If you set this yourself then you can reload classes even if fields and methods have been added

•If the changes are incompatable (e.g. change of a variable name) it will still throw an exception



Version Control Example

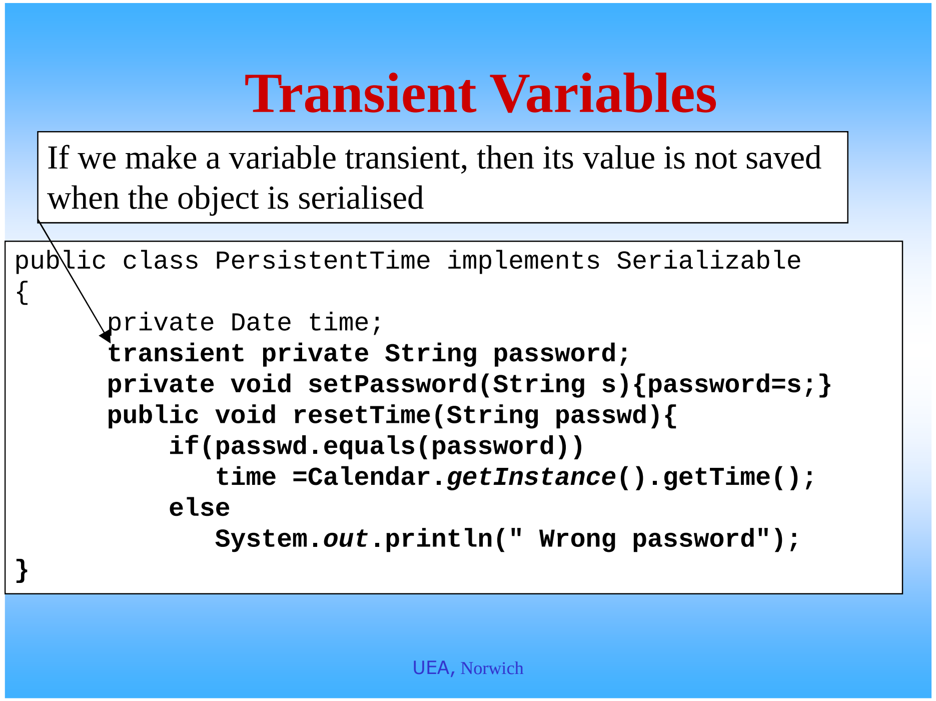
The value is unimportant

```
public class PersistentTime implements Serializable
{
      static final long serialVersionUID = 101L;
```

• The *serialVersionUID* is saved in the binary file with the object information

•If we now make minor changes to our class, (e.g. add a field or method) we can still load old persistent objects.

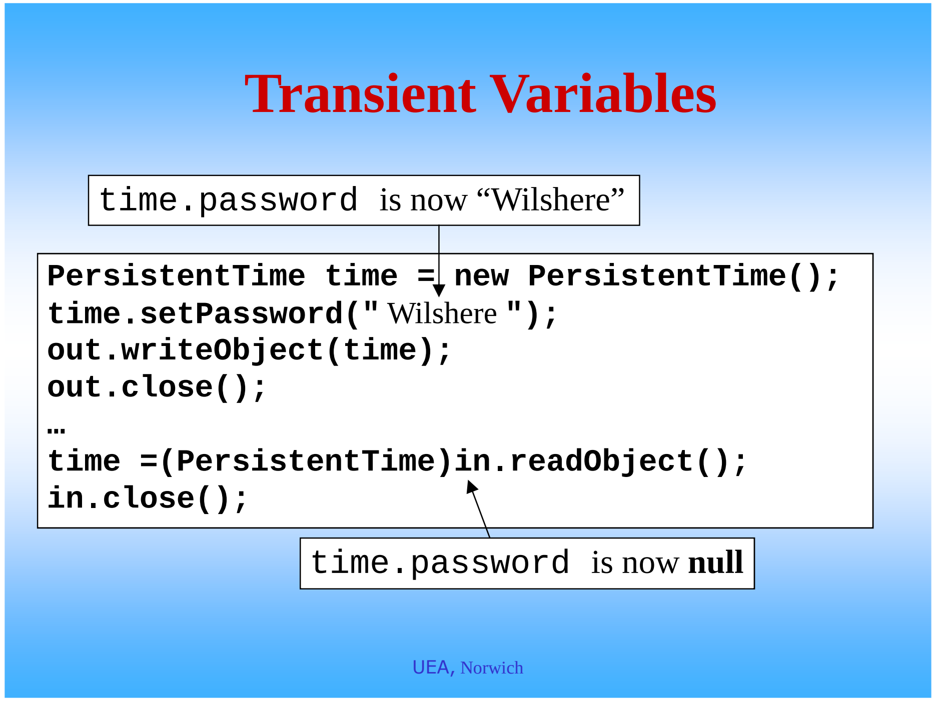
•If we make major changes to our class (e.g. change data structures or variable names) we can change the UID to identify problems with persistent objects being incorrectly loaded



Transient Variables

If we make a variable transient, then its value is not saved when the object is serialised

```
public class PersistentTime implements Serializable
{
      private Date time;
      transient private String password;
      private void setPassword(String s){password=s;}
      public void resetTime(String passwd){
          if(passwd.equals(password))
             time =Calendar.getInstance().getTime();
          else
             System.out.println(" Wrong password");
}
```

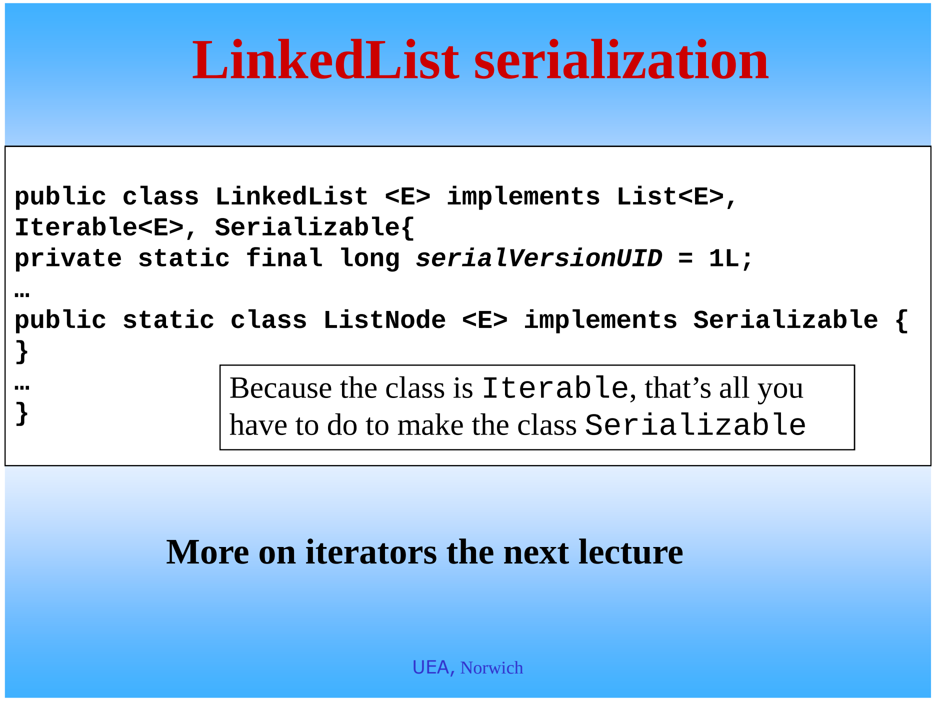


Transient Variables

`time.password` is now “Wilshere”

```
PersistentTime time = new PersistentTime();
time.setPassword(" Wilshere ");
out.writeObject(time);
out.close();
…
time =(PersistentTime)in.readObject();
in.close();
```

`time.password` is now **null**



LinkedList serialization

```
public class LinkedList <E> implements List<E>,
Iterable<E>, Serializable{
private static final long serialVersionUID = 1L;
…
public static class ListNode <E> implements Serializable {
}
…
}
```

Because the class is `Iterable`, that’s all you have to do to make the class `Serializable`

More on iterators the next lecture

LinkedList serialization

Saves file to bytecode

```
String filename = "list.ser";
LinkedList<String> l = new LinkedList<String>();
l.add(“Walcott"); l.add(“Wilshere");
FileOutputStream fos = new FileOutputStream(filename);
ObjectOutputStream out = new ObjectOutputStream(fos);
out.writeObject(l);
out.close();
l.add(“Ozil");
FileInputStream fis = new FileInputStream(filename);
ObjectInputStream in = new ObjectInputStream(fis);
LinkedList<String> l2=(LinkedList<String> )in.readObject();
in.close();
```

loads file from bytecode

l=

```
Walcott
Wilshere
Ozil
```

l2=

```
Walcott
Wilshere
```

Customised Serialisation

We can control how an object is serialised by implementing the following two methods

```
private void writeObject(ObjectOutputStream out)
throws IOException;

private void readObject(ObjectInputStream in)
throws IOException, ClassNotFoundException
```

- We still call `readObject` and `writeObject` as before, but now the JVM checks whether they have been implemented and if so, calls them
- Note they must be private so they cannot be over-ridden

Customised Serialisation

The table in a HashMap is declared transient. We do not want to store all the empty entries in the table

```
public class HashMap<K,V> extends AbstractMap<K,V>
implements Map<K,V>, Cloneable, Serializable
{
transient Entry[] table;
private void writeObject(java.io.ObjectOutputStream s)
         throws IOException
     {
              s.defaultWriteObject();
```

This will write all the non-transient data such as load factor and threshold using the normal method

HashMap serialization

- The method can then write out any appropriate transient data in the required format

Write out number of buckets

Write out the size (number of Mappings)

This returns a set view of the map then iterates over the elements

```
s.writeInt(table.length);

s.writeInt(size);


Iterator<Map.Entry<K,V>>
      i = (size > 0) ? entrySet().iterator() : null;
      if (i != null) {
            while (i.hasNext()) {
                  Map.Entry<K,V> e = i.next();
                  s.writeObject(e.getKey());
                  s.writeObject(e.getValue());
            }
      }
```

Your Own Serialization Protocol

- the java `Externalizable` Interface allows you to completely define how to serialize with use of `getObject` or `setObject`

```
public void writeExternal(ObjectOutput out)
            throws IOException;

public void readExternal(ObjectInput in)
            throws IOException, ClassNotFoundException;
```

For example, if you know how to write and read PDF (the sequence of bytes required), you could provide the PDF-specific protocol in the `writeExternal` and `readExternal` methods

Caching

We cannot overwrite an object once saved without closing and reopening the stream

Time here say 00:00

```
 out.writeObject(time);
//Do something that takes ages

out.writeObject(time);
FileInputStream fis = new FileInputStream(filename);
ObjectInputStream in = new ObjectInputStream(fis);
time =(PersistentTime)in.readObject();
```

Time here now 00:30

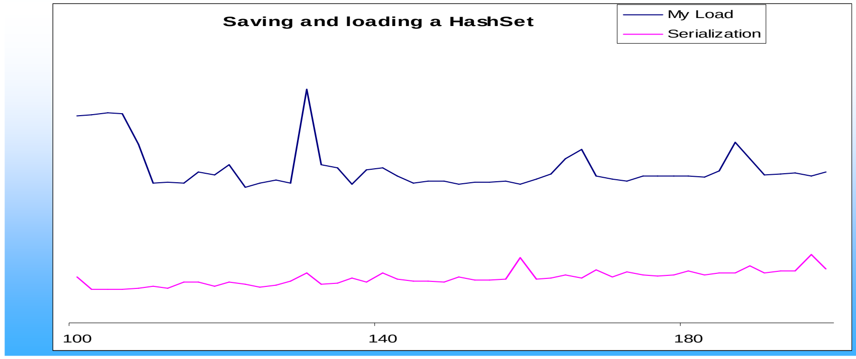
The time that persists is in fact 00:00, even though we have written again

Solution:

1. Always close and reopen
2. Flush the cache by calling `out.reset()`

Performance

- For basic classes, serialisation may actually be slower than writing bespoke save file methods
- For more complex classes the provided serialisation is generally better



Reflection

Reflection is the ability of a class or object to examine itself

Confucius says:
“By three methods we may learn wisdom: first, by **reflection,** which is noblest**;** second, by imitation, which is easiest; and third, by experience, which is the most bitter.”

Reflection

Reflection is the ability of a class or object to examine itself

```
public void someMethod(Object o) {


}
```

What is o? It could be anything at all, a String, a BinaryTree, an Array etc

Reflection allows you to dynamically find out information about

1. **The data fields**
2. **The methods**
3. **The constructors**

Tools for reflecting on an object are available in java.lang.reflect

the class `Class`

The class `Class` contains the tools for reflection

```
public void someMethod(Object o){
     Class c = o.getClass();
System.out.println(“Class=”+c.getName());
}
 public static void main(String[] args){
  someMethod("This is a string");
  someMethod(new Integer(0));
  someMethod(new Deck());
}
```

Class =java.lang.String
Class =java.lang.Integer
Class =Deck

the class `Class`

We can get all the info available from the `Class` object

```
public void someMethod(Object o){
     Class c = o.getClass();
     Method[] m = c.getDeclaredMethods();
     Field[] f=c.getDeclaredFields();
}
```

Returns the public methods and fields of the object

This and a lot more can all be found via reflection.

Calling methods

- We can not only inspect the object, we can do things to it!
- We can also create new objects of the same type

```
public static void someMethod(Object o)
{
      Class c = o.getClass();
      Method method=c.getDeclaredMethods()[0];
      try{
            Object obj=c.newInstance();
            method.invoke(obj);
   System.out.println(" Method name is“ +method.getName()+
   "return is ="+method.invoke(obj));
      }catch(Exception e){
      System.out.println(" Exception = "+e);}
}
```

Calling methods

```
public static void main(String[] args)
{
        someMethod("This is a string");
        someMethod(new Deck());
}
```

```
Output
Method name is hashCode return is =0
Method name is deal return is =ACE of DIAMONDS
```

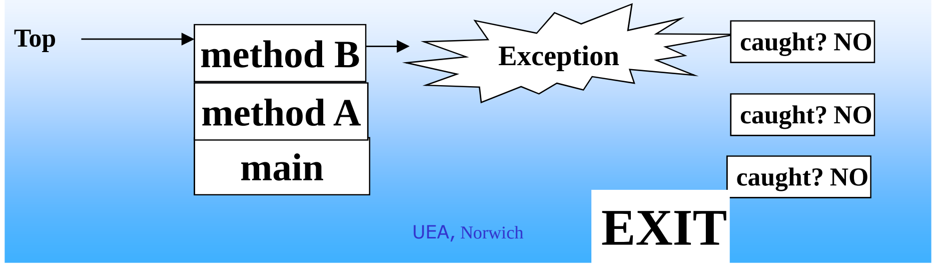
That’s just the basics, there is a lot more you can do with reflection

Exceptions

Exceptions are “exceptional events” in your program that disrupt the normal flow of execution

Exceptions

1. Execution of statements stops at the exact point the Exception occurred
2. An Exception object is created with information about the event
3. The exception is “thrown” down the method stack until it is either “caught” or the program terminates



Built in Exceptions

All built in Exceptions inherit from the `Exception` class in `java.lang`

http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Exception.html

`ArithmeticException` –

Thrown when an exceptional arithmetic condition has occurred. For example, an integer "divide by zero" throws an instance of this class.

```
int x=10,y=0,z;
int z=x/y;
```

Exception in thread "main" java.lang.ArithmeticException: / by zero

Note this is not thrown by double divide by zero

```
double x=10,y=0,z;
z=x/y;
```

z = “infinity”

Built in Exceptions

`ArrayIndexOutOfBoundsException`

Thrown if a program attempts to access an index of an array that does not exist

```
      int[] a = new int[10];
            for(int i=0;i<=a.length;i++)
                  a[i]=i*i;
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
```

`NullPointerException`

Thrown when an application attempts to use null in a case where an object is required.

```
      String str=null;
      String str2=“A String”;
      if(str.equals(str2))
            str=“FC”;
```

```
Exception in thread "main" java.lang.NullPointerException
```

And all the rest!

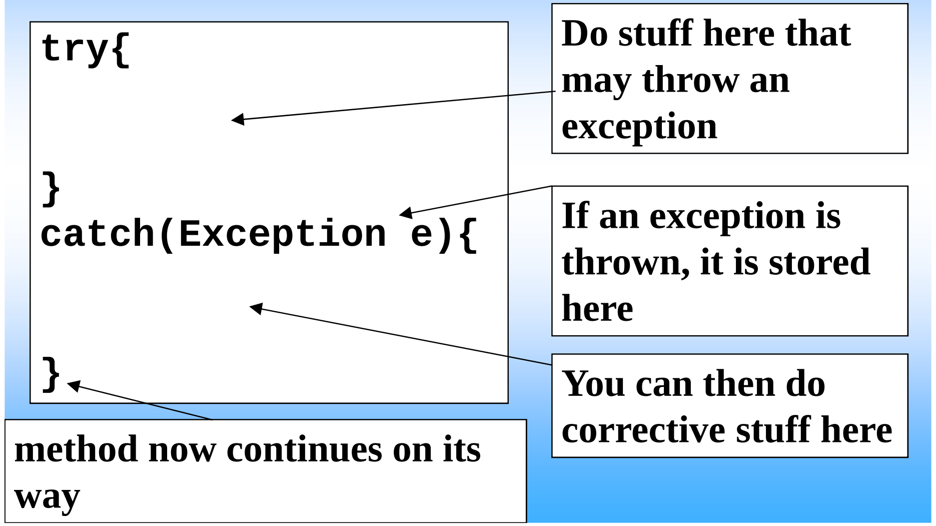
Built in Exceptions

Direct Known Subclasses:

AnnotationTypeMismatchException, ArithmeticException, ArrayStoreException, BufferOverflowException, BufferUnderflowException, CannotRedoException, CannotUndoException, ClassCastException, CMMException, ConcurrentModificationException, DOMException, EmptyStackException, EnumConstantNotPresentException, EventException, IllegalArgumentException, IllegalMonitorStateException, IllegalPathStateException, IllegalStateException, ImagingOpException, IncompleteAnnotationException, IndexOutOfBoundsException, JMRuntimeException, LSException, MalformedParameterizedTypeException, MissingResourceException, NegativeArraySizeException, NoSuchElementException, NullPointerException, ProfileDataException, ProviderException, RasterFormatException, RejectedExecutionException, SecurityException, SystemException, TypeNotPresentException, UndeclaredThrowableException, UnmodifiableSetException, UnsupportedOperationException

`try … catch`

The exception handling mechanism in java is the `try…catch…finally` syntax



```
try{


}
catch(Exception e){


}
```

Do stuff here that may throw an exception

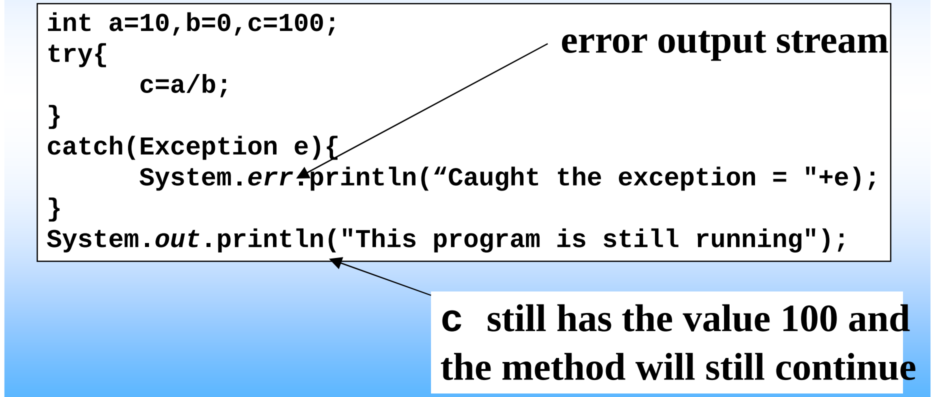
If an exception is thrown, it is stored here

You can then do corrective stuff here

method now continues on its way

Catching Exceptions

Catching an exception gives us the option of carrying on with the execution of the program

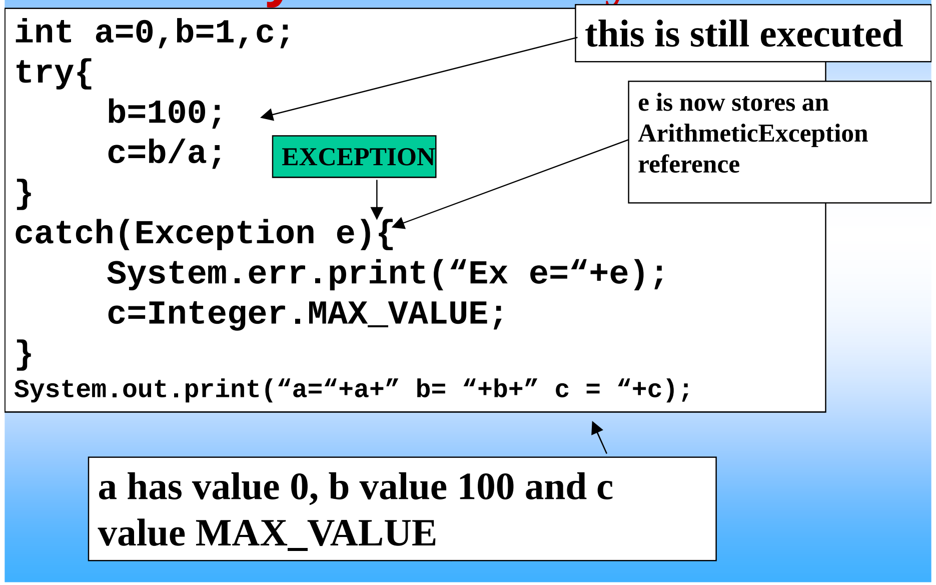


```
int a=10,b=0,c=100;
try{
      c=a/b;
}
catch(Exception e){
      System.err.println(“Caught the exception = "+e);
}
System.out.println("This program is still running");
```

error output stream

`c` still has the value 100 and the method will still continue

try … catch syntax



```
int a=0,b=1,c;
try{
      b=100;
      c=b/a;
}
catch(Exception e){
      System.err.print("Ex e="+e);
      c=Integer.MAX_VALUE;
}
System.out.print("a="+a+" b= "+b+" c = "+c);
```

this is still executed

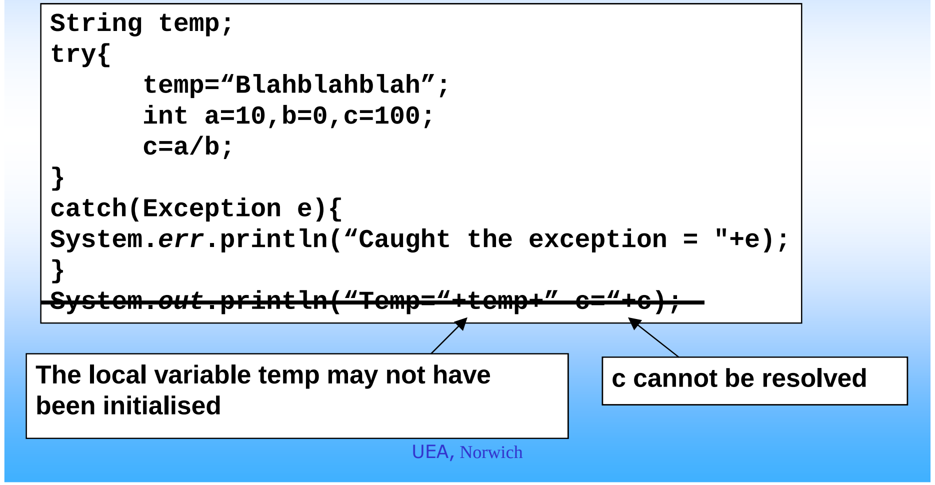
EXCEPTION

e is now stores an ArithmeticException reference

a has value 0, b value 100 and c value MAX_VALUE

try…catch declarations

Variables declared within a try or catch have scope limited to block



```
String temp;
try{
      temp="Blahblahblah";
      int a=10,b=0,c=100;
      c=a/b;
}
catch(Exception e){
System.err.println("Caught the exception = "+e);
}
~~System.out.println("Temp="+temp+" c="+c);~~
```

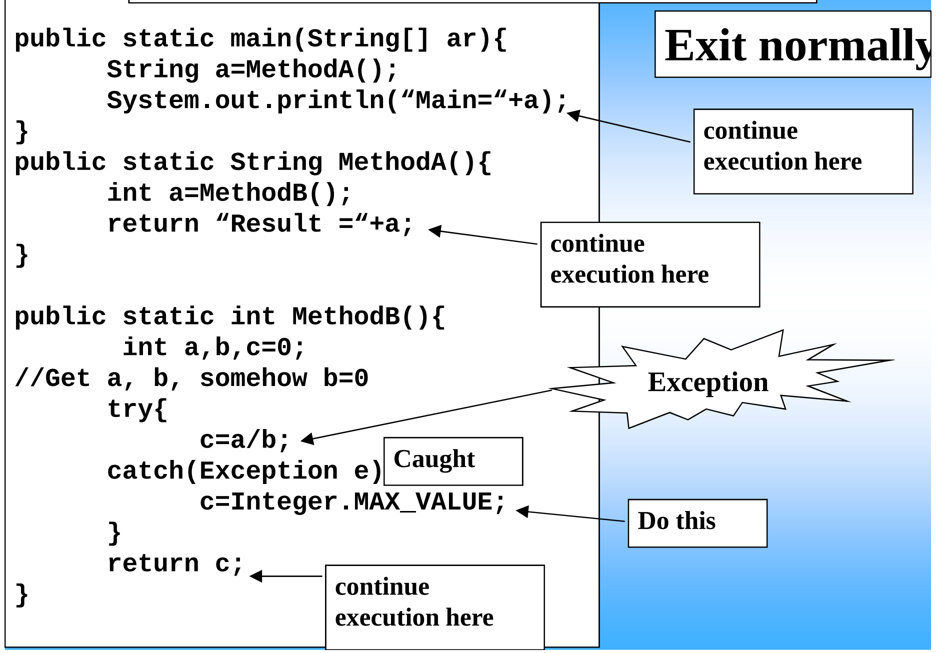
The local variable temp may not have been initialised

c cannot be resolved

Details about Exceptions

1. We can catch the exception anywhere up the stack
2. We can explicitly catch different types of exceptions
3. `finally` block always executes
4. Exceptions can be checked or unchecked

1. We can catch the exception anywhere up the stack



```
public static main(String[] ar){
      String a=MethodA();
      System.out.println("Main="+a);
}
public static String MethodA(){
      int a=MethodB();
      return "Result ="+a;
}

public static int MethodB(){
       int a,b,c=0;
//Get a, b, somehow b=0
      try{
            c=a/b;
      catch(Exception e)
            c=Integer.MAX_VALUE;
      }
      return c;
}
```

Exit normally

continue execution here

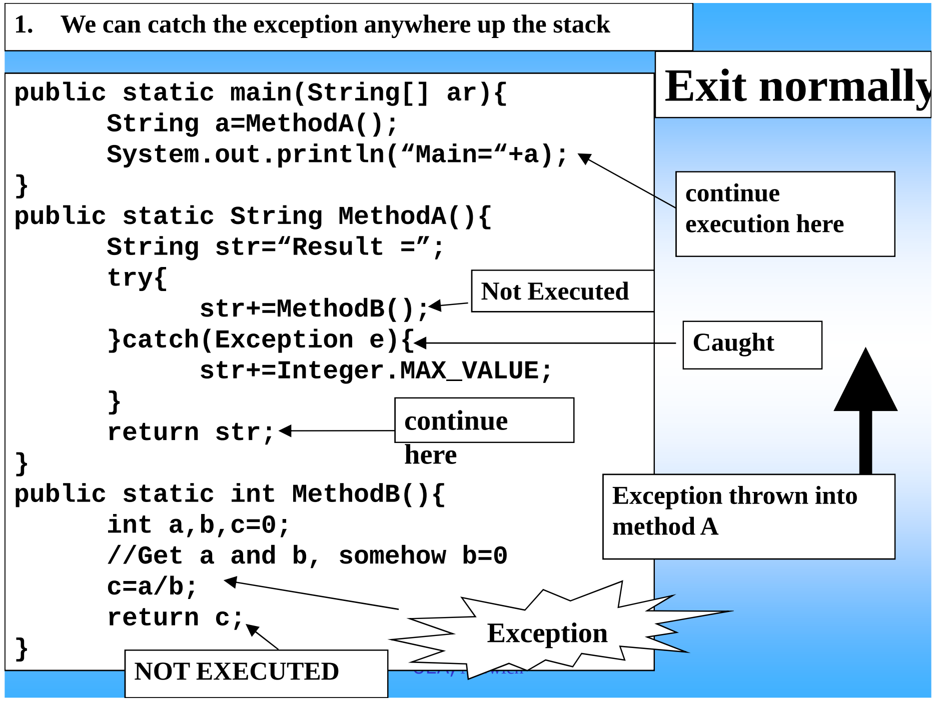
continue execution here

Exception

Caught

Do this

continue execution here



1. We can catch the exception anywhere up the stack

```
public static main(String[] ar){
      String a=MethodA();
      System.out.println(“Main=“+a);
}
public static String MethodA(){
      String str=“Result =”;
      try{
            str+=MethodB();
      }catch(Exception e){
            str+=Integer.MAX_VALUE;
      }
      return str;
}
public static int MethodB(){
      int a,b,c=0;
      //Get a and b, somehow b=0
      c=a/b;
      return c;
}
```

Exit normally

continue execution here

Not Executed

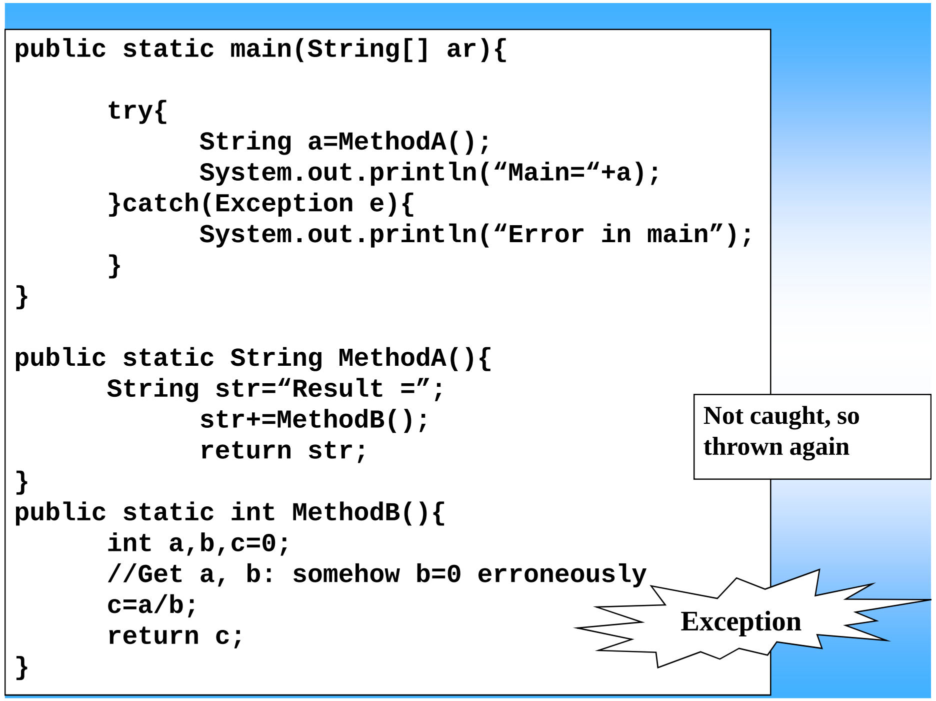
Caught

continue here

Exception thrown into method A

Exception

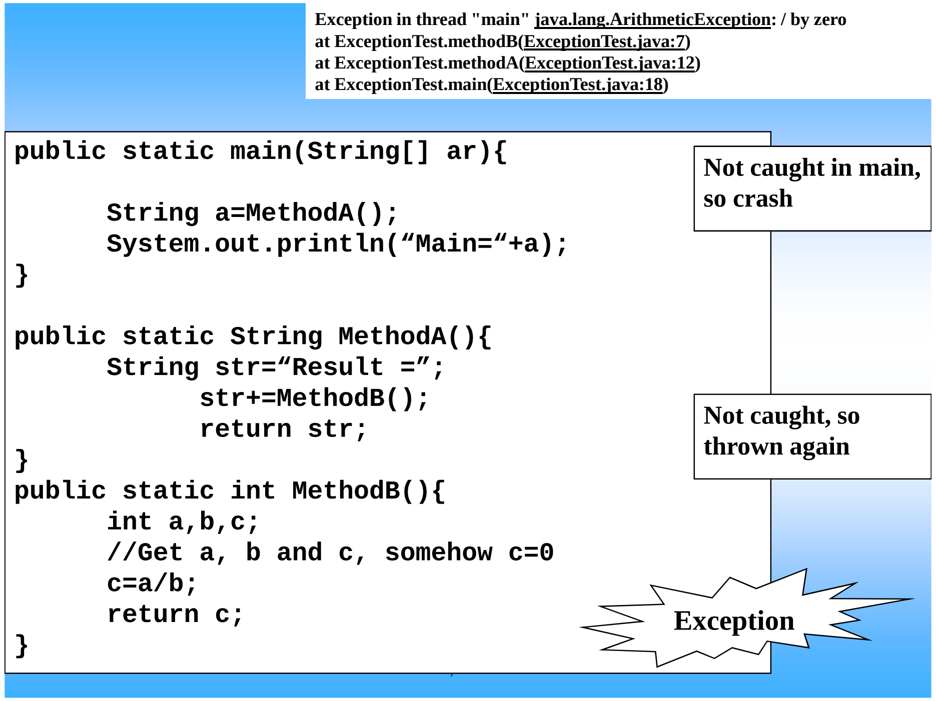
NOT EXECUTED



```
public static main(String[] ar){

      try{
            String a=MethodA();
            System.out.println(“Main=“+a);
      }catch(Exception e){
            System.out.println(“Error in main”);
      }
}

public static String MethodA(){
      String str=“Result =”;
            str+=MethodB();
            return str;
}
public static int MethodB(){
      int a,b,c=0;
      //Get a, b: somehow b=0 erroneously
      c=a/b;
      return c;
}
```

Not caught, so thrown again

Exception



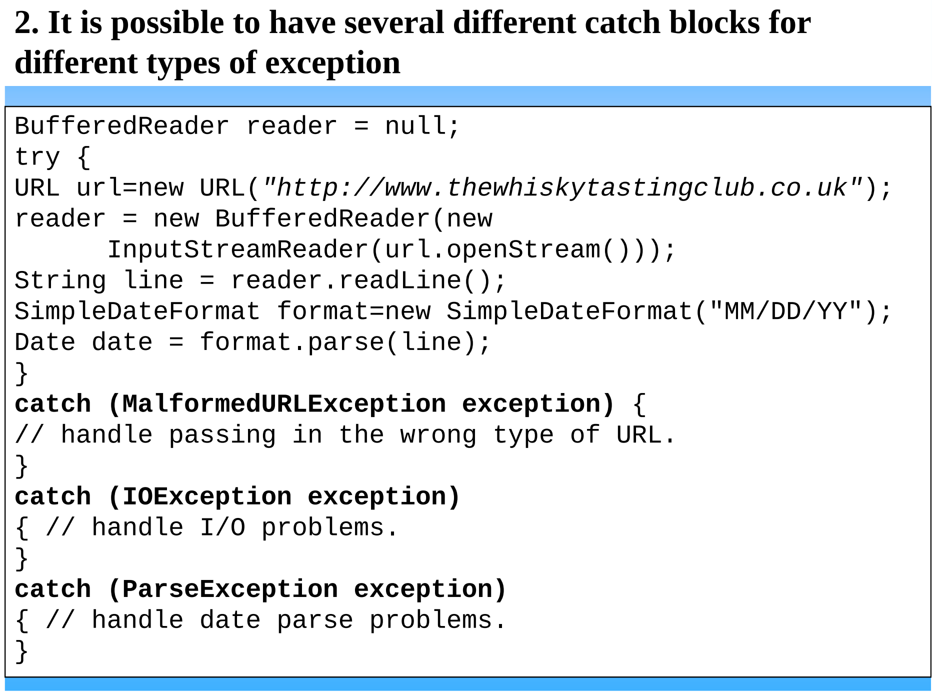
Exception in thread "main" java.lang.ArithmeticException: / by zero
at ExceptionTest.methodB(ExceptionTest.java:7)
at ExceptionTest.methodA(ExceptionTest.java:12)
at ExceptionTest.main(ExceptionTest.java:18)

```
public static main(String[] ar){

      String a=MethodA();
      System.out.println(“Main=“+a);
}

public static String MethodA(){
      String str=“Result =”;
            str+=MethodB();
            return str;
}
public static int MethodB(){
      int a,b,c;
      //Get a, b and c, somehow c=0
      c=a/b;
      return c;
}
```

Not caught in main, so crash

Not caught, so thrown again

Exception



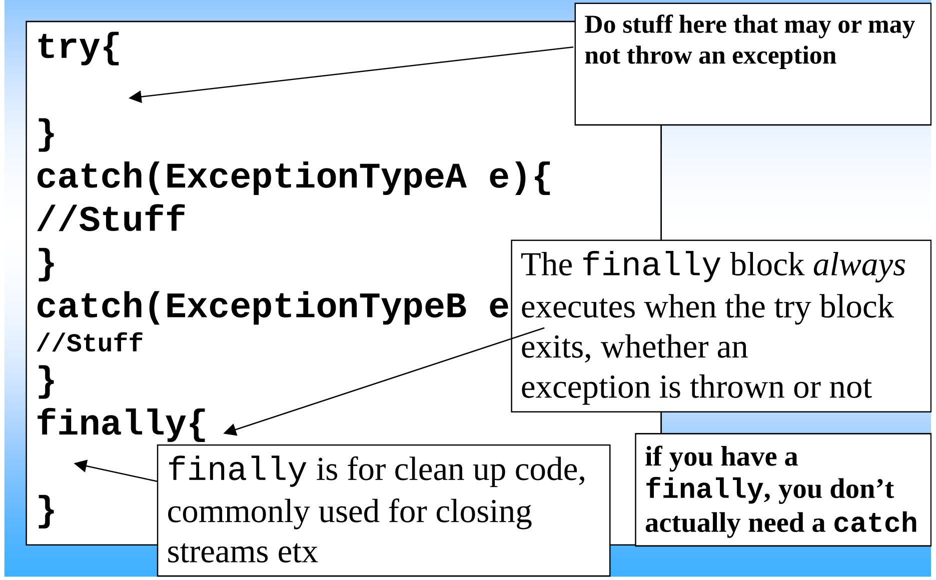
2. It is possible to have several different catch blocks for different types of exception

```
BufferedReader reader = null;
try {
URL url=new URL("http://www.thewhiskytastingclub.co.uk");
reader = new BufferedReader(new
      InputStreamReader(url.openStream()));
String line = reader.readLine();
SimpleDateFormat format=new SimpleDateFormat("MM/DD/YY");
Date date = format.parse(line);
}
catch (MalformedURLException exception) {
// handle passing in the wrong type of URL.
}
catch (IOException exception)
{ // handle I/O problems.
}
catch (ParseException exception)
{ // handle date parse problems.
}
```

2. In Java SE 7 onwards, it is possible to group Exceptions

```
BufferedReader reader = null;
try {
URL url=new
URL("http://www.thewhiskytastingclub.co.uk");
reader = new BufferedReader(new
      InputStreamReader(url.openStream()));
String line = reader.readLine();
SimpleDateFormat format=new
SimpleDateFormat("MM/DD/YY"); Date date =
format.parse(line);
}
catch (MalformedURLException exception) {
// handle passing in the wrong type of URL.
}
catch (IOException exception | ParseException exception)
{ // handle I/O problems and Parse problems.
}
```

try … catch … finally



```
try{

}
catch(ExceptionTypeA e){
//Stuff
}
catch(ExceptionTypeB e
//Stuff
}
finally{

}
```

Do stuff here that may or may not throw an exception

The `finally` block *always* executes when the try block exits, whether an exception is thrown or not

`finally` is for clean up code, commonly used for closing streams etx

if you have a `finally`, you don't actually need a `catch`

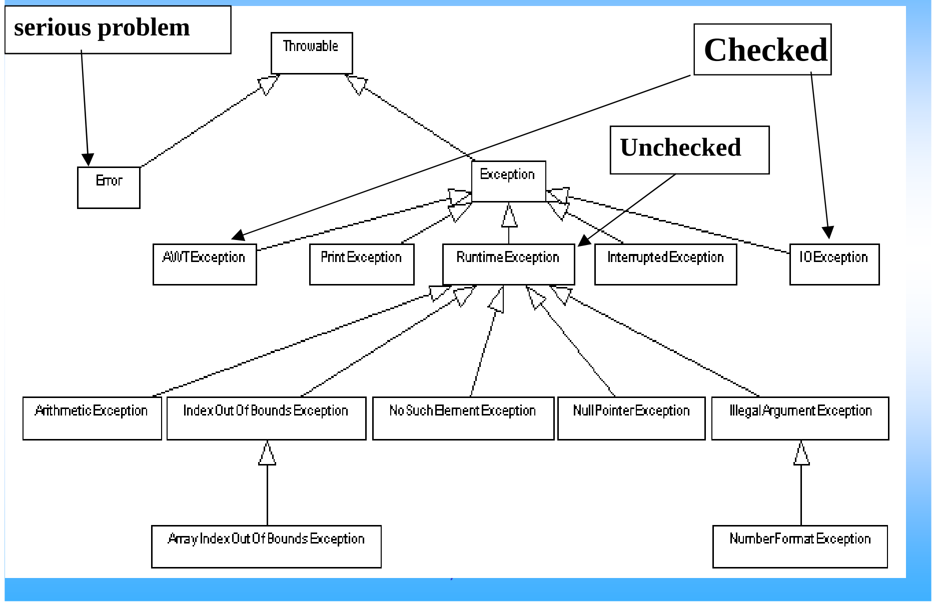
Finally example

```
PrintWriter out = null;

try {
out = new PrintWriter(new FileWriter("OutFile.txt"));

//Do stuff
}
catch (IOException|SQLException ex) {
// handle I/O problems and Parse problems.

}
finally {
//Clean up code, close file
      if (out != null)
            out.close();

}
```

Checked vs Unchecked Exceptions

- The Exceptions we have seen are unchecked.
- This means we don't have to put calls to code thatmight generate them within a `try` … `catch` block
- Exceptions inheriting from `RunTimeException` are by default unchecked
- Checked Exceptions force the user to `catch`
- We can make any Exception checked by including `throws Exception`

Checked vs Unchecked Exceptions



serious problem

Checked

Unchecked

Writing your own Exceptions

Extending `Exception` means this will be a checked Exception

```
public class InsufficientFundsException extends
Exception{
   private double amount;
   public InsufficientFundsException(double amount)
   {
      this.amount = amount;
   }
   public double getAmount()
   {
      return amount;
   }

}
```

Writing your own Exceptions

Extending `RuntimeException` means this will be an unchecked Exception

```
public class InvalidInputException extends
RunTimeException
{
   private double amount;
   public InvalidInputException (double amount){
      this.amount = amount;
   }

   public double getAmount(){
      return amount;
   }
```

Writing your own Exceptions

```
 public class CurrentAccount{
   private double balance;
   public CurrentAccount(){
   }
   public void deposit(double amount){
       if(amount<0)
           throw new InvalidInputException(amount);
      balance += amount;
   }
   public void withdraw(double amount) throws
InsufficientFundsException {
      if(amount <= balance){
         balance -= amount;
      }
      else{
         double needs = amount - balance;
         throw new InsufficientFundsException(needs);
      }
   }
```

Extends `RuntimeException`, so no `throws` required

Extends `Exception`, so `throws` required, or it wont compile

Advantages of using Exceptions

- To handle exceptions, we can either return special error codes or use the built in Exceptions
- The benefits of using Exceptions are

1. We can separate Error-Handling Code from Application Code

2. We can propagate exceptions up the call stack and thus deal with them in the appropriate place

3. We can use inheritance to group types of exception thus increasing the information we can convey

1. Separating Error-Handling Code from Application Code

```
readFile {
open the file;
determine its size;
allocate that much memory;
read the file into memory;
close the file;
}
```

Error checking for input can massively clutter up your code

```
errorCodeType readFile {
initialize errorCode = 0;
open the file;
if (theFileIsOpen) {
determine the length of the file;
if (gotTheFileLength) {
            if (gotEnoughMemory) {
            read the file into memory;
                if (readFailed) {
                        errorCode = -1;
                }
            }
            else {
            errorCode = -2;
            }
} else { errorCode = -3; }
            return errorCode; }
```

1. Separating Error-Handling Code from Application Code

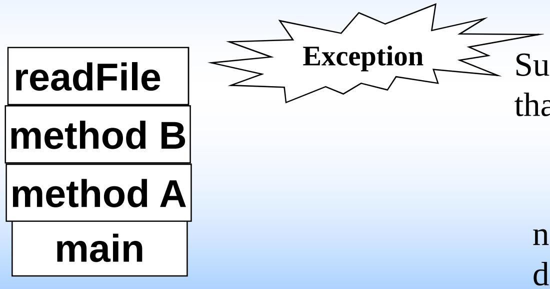
Exceptions make it much cleaner

```
readFile {
open the file;
determine its size;
allocate that much memory;
read the file into memory;
}
```

```
readFile {
open the file;
determine its size;
allocate that much memory;
read the file into memory;
}
catch (fileOpenFailed) {
            doSomething;
}catch
(sizeDeterminationFailed) {
            doSomething;
} catch
(memoryAllocationFailed) {
            doSomething;
}
```

Advantage 2: Propagating Errors Up the Call Stack

With Exceptions we can deal with the error at the place it matters



Suppose exceptions that occur here

needs to be dealt with here

Propagating Errors Up the Call Stack

Exceptions may occur here

```
methodB {
call readFile;
}
 methodB {
call methodA;
}

main {
call methodB;
}
```

With error codes we need to return a lot of values

```
errorCodeType methodB {
        errorCodeType error; error = call readFile;
        if (error) return error;
        else proceed;
}
errorCodeType methodA {
        errorCodeType error;
        error = call methodB;
        if (error) return error;
        else proceed;
}
main {
        errorCodeType error;
        error = call methodA;
        if (error) doErrorProcessing;
}
```

Propagating Errors Up the Call Stack

Exceptions make it easier

No clutter in these methods

```
methodB {
call readFile;
}
 methodB {
call methodA;
}

main {
call method2;
}
```

```
methodB {
        call readFile;
        }
methodA {
        call methodB;
}

main {
        try{
                call methodA;
        }
        catch(Exception){
        doErrorProcessing;
        }
```

Using Exceptions 1: Data Entry

```
static public void loadFile()
{
System.out.println("Enter a file name");
BufferedReader buf_reader = new BufferedReader (new
InputStreamReader (System.in));
BufferedReader in=null;

try{
in= new BufferedReader(new buf_reader.readLine ()));
}catch(Exception e){
System.out.println("UNKNOWN FILE NAME");
loadFile();

}
```

Using Exceptions 2: Debugging

Exception caught =java.io.FileNotFoundException: C:\Research\Code\WekaTest\PlayGolfTra.arff

```
Instances train;
try{
      r= new FileReader(str1);
      train = new Instances(r);
      IB1 knn=new IB1();
      knn.buildClassifier(train);
}
catch(Exception e){
      System.out.println(" Exception caught ="+e);
}
```

Exception caught =weka.core.UnassignedClassException: Class index is negative (not set)!

Summary

1. Exceptions are “exceptional events” in your program that disrupt the normal flow of execution
2. Exceptions are thrown with the keyword `throws`
3. Exceptions are caught in `try` … `catch` … `finally` blocks.
4. Exceptions can be checked or unchecked. Checked exceptions are explicitly returned with the reserved word `throw`
5. Exceptions can be propagated up the stack and caught at any point
6. Exceptions can separate Error-Handling Code from Application Code