CM 10227: Lecture 9

Dr Rachid Hourizi and Dr Michael Wright

November 29, 2016

Resources

- More help with this course
  - Moodle
  - E-mail - programming1@lists.bath.ac.uk
- Online C and Java IDE
  - https://www.codechef.com/ide
  - Remember to select Java from the drop down menu.
- Books
  - Java by Dissection (Free pdf online)
  - The Java Tutorial (https://docs.oracle.com/javase/tutorial/)

Resources

- The places that you can get additional support if you are finding the pace of the course a little fast now include
  - A labs (Continued from week 1)
  - B labs
  - ... Wednesday 11:15-13:05 EB0.7
  - ... Fridays 17:15 to 19:15 in CB 5.13)
  - The Drop in Session
  - ... booked 20 min appointments
  - ... Friday 11.15-13.05 1E 3.9
  - PAL sessions (Mondays 14:15 to 15:05 1E 3.9)

Last Week

- Interfaces
- Abstract Classes

This week

- Errors
- Exceptions
- Style : Writing Better Code

Overview

- Defensive programming.
- Anticipating that things could go wrong.
- Exception handling and throwing.
- Error reporting.

Robustness of Code

- Robustness is the ability of a computer system to...

- ... cope with errors during execution
- ... and cope with erroneous input.

Example Programming Errors

- Some problems really should be addressed by the programmer during coding
- Incorrect implementation.
  - does not meet the specification.
- Inappropriate object request.
  - e.g. invalid index.

Guarding

- Some of those problems can, in turn be addressed with appropriate conditional statements

```
if (number is above zero){
    calculate corresponding Fibonacci number
}
```

```
if (index is within bounds){
    return element with that index from Array
}
```

Guarding

- This kind of conditional statement that is used to decide whether a branch of the program will execute is called a guard
- Strictly, a guard is the boolean expression that must evaluate to true if the 'guarded' block of code is to execute
- You should certainly use guards in your programs

- (Arguably), however, not all errors are programmer errors
- Errors often arise from the environment:
  - Incorrect URL entered.
  - Network interruption.
- File processing is particular error-prone:
  - Missing files.
  - Lack of appropriate permissions.

Defensive Programming

- Client-server interaction.
  - Should a server assume that clients are well-behaved?
  - Or should it assume that clients are potentially hostile?
- Significant differences in implementation required.

Issues to be Addressed

- How much checking by a server on method calls?
- How to report errors?
- How can a client anticipate failure?
- How should a client deal with failure?

An example: The AddressBook Project

- Create an Online AddressBook object
- Try to remove an entry.
- A runtime error results.
- Whose fault is this?

An example: The AddressBook Project

- Anticipation and prevention are preferable to apportioning blame.

Arguments And Errors

- Arguments represent a major vulnerability for a server object.
- Constructor arguments initialize state.
- Method arguments often contribute to behaviour.
- Argument checking is one defensive measure.

Checking the Key

```java
public void removeDetails(String key) {
  if (keyInUse(key)) {
    ContactDetails details = book.get(key);
    book.remove(details.getName());
    book.remove(details.getPhone());
    numberOfEntries --;
  }
}
```

Aside : i - - and - - i

- Note the use of the decrement operator on the previous slide (double minus or - -)
- The use of that decrement operator after the variable (numberOfEntries) means decrement the value **after** any operations involving that variable have concluded
- Putting the decrement operator **before** the variable name (- -i) would mean decrement and then perform those operations.

- Similarly, the increment operator (++) can be used before the variable (increment then use, ++i) or afterwards (i++, use then increment)

Aside : i - - and - - i

```
public static void main (String[] args) {
    int i = 3;
    i++;
    System.out.println(i);
    ++i;
    System.out.println(i);
    System.out.println(++i);
    System.out.println(i++);
    System.out.println(i);
}
```

what is printed ?

Aside : i - - and - - i

```
4
5
6
6
7
```

Aside : i - - and - - i

```java
public static void main (String[] args) {
    int i = 3;
    i++;
    System.out.println(i); // prints 4
    ++i;
    System.out.println(i);  // prints 5

    // increments before the print so prints 6
    System.out.println(++i);

    // increments *after* the print so prints 6
    System.out.println(i++);

    System.out.println(i); // prints 7
}
```

Server Error Reporting Questions:

- How to report illegal arguments?
- To the user?
  - Is there a human user?
  - Can they solve the problem?
- To the client object?
  - Return a diagnostic value?
  - Throw an exception?

Returning a diagnostic

```java
public boolean removeDetails(String key) {
  if (keyInUse(key)) {
    ContactDetails details =
          (ContactDetails)book.get(key);
    book.remove(details.getName());
    book.remove(details.getPhone());
    numberOfEntries--;
    return true;
  }
  else{
    return false;
  }

}
```

Client Responses

- How should the Client Respond?
- Test the return value?
  - Attempt recovery on error.
  - Avoid program failure.
- Ignore the return value?
  - Means Error cannot be prevented.
  - Likely to lead to program failure.
- Exceptions are preferable.

Exception Throwing Principles

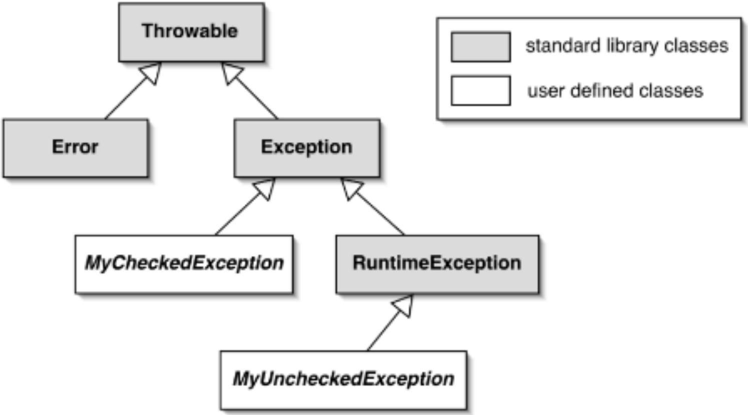
- A special language feature.
- No special return value needed.
- Errors cannot be ignored in the client.
  - The normal flow-of-control is interrupted.
- Specific recovery actions are encouraged.

Throwing an Exception

```
public ContactDetails getDetails (String key){
  if(key == null){
    throw new NullPointerException( "null␣key␣in␣
        getDetails␣");
  }
  return book.get(key);
}
```

Throwing an Exception

- An exception object is constructed:
  - new ExceptionType("...");
- The exception object is thrown:
  - throw ...
- Javadoc documentation:
  - @throws ExceptionType reason



Throwable

Error

Exception

MyCheckedException

RuntimeException

MyUncheckedException

▨ standard library classes

□ user defined classes

java.lang.Error

- Indicates serious problems that a reasonable application should not try to catch
- A method is not required to declare in its throws clause any subclasses of Error

Checked Exceptions - java.lang.Exception

- Checked exceptions Subclass of Exception
  - Use for anticipated failures.
  - Where recovery may be possible.

- Checked exceptions must be caught in your code
  - compiler enforces it
- e.g. readLine() throws a IOException which must be caught

The Effect of An Exception

- The throwing method finishes prematurely.
- No return value is returned.
- Control does not return to the clients point of call.
  - So the client cannot carry on regardless.
- A client may catch an exception.

Unchecked Exceptions - java.lang.RuntimeException

- Use of these is unchecked by the compiler.
- Cause program termination if not caught.
  - This is the normal practice.
- IllegalArgumentException is a typical example.

Runtime Exception

- The next question might be:
- If it's so good to document a method's API, including the exceptions it can throw, why not specify runtime exceptions too?
- Runtime exceptions represent problems that are the result of a programming problem, and as such, the API client code cannot reasonably be expected to recover from them or to handle them in any way.
- Such problems include:
  - arithmetic exceptions, such as dividing by zero
  - pointer exceptions, such as trying to access an object through a null reference
  - and indexing exceptions, such as attempting to access an array element through an index that is too large or too small.

Argument Checking

```
public ContactDetails getDetails (String key){
  if(key == null){
    throw new NullPointerException
        ("null␣key␣in␣getDetails");
  }

  if(key.trim().length() == 0){
    throw new IllegalArgumentException
         ("Empty␣key␣passed␣to␣getDetails");
   }

  return book.get(key);
}
```

Preventing Object Creation

```
public ContactDetails newDetails (String name){
  if(name == null){
    throw new NullPointerException
        ("null␣name␣in␣newDetails");
  }

  if(name.trim().length() == 0){
    throw new IllegalArgumentException
         ("Empty␣name␣passed␣to␣newDetails");
   }

  return new ContactDetails(name);
}
```

Exception Handling

- Checked exceptions are meant to be caught.
- The compiler ensures that their use is tightly controlled.
  - In both server and client.
- Used properly, failures may be recoverable.

The Throws Clause

- Methods throwing a checked exception must include a throws clause:

```
public void saveToFile(String destinationFile)
        throws IOException
```

The Try Block

- Clients catching an exception must protect the call with a try block:

```
try{
  Protect one or more statements here .
}
catch(IOException e){
  Report and recover from  exception here.
}
```

The Try Block

```
try{
  addressbook.saveToFile(filename);
  tryAgain = false;
}
catch(IOException e) {
  System.out.println("Unable␣to␣save␣to␣" + filename);
  tryAgain = true;
}
```

Checking Multiple Exceptions

```java
try{
  ...
  ref.process() ;
  ...
}
catch (EOFException e){
  // Take action on an end-of-file exception
  ...
}
catch(FileNotFoundException e) {
  //Take action on a file-not-found exception
   ...
}
```

Finally

```
try{
  Protect one or more statements here .
}
catch(Exception e){
  Report and recover from  exception here.
}
finally{
  Perform any actions here that must occur
  whether or not an exception is thrown
}
```

The Finally Clause

- A finally clause is executed even if a return statement is executed in the try or catch clauses.
- A uncaught or propagated exception still exits via the finally clause.

FileWriter: an Example

- Use the FileWriter class.
  - Open a file.
  - Write to the file.
  - Close the file.
- Failure at any point results in an IOException.

```
try {
  FileWriter writer = new FileWriter(name of file );
  while ( there is more text to write ){
    ...
    writer.write(next piece of text)
    ...
}
catch(IOException e){
  something went wrong with accessing the file
}
finally{
  writer.close();
}
```

Defining New Exceptions

- Extend Exception or RuntimeException.
- Define new types to give better diagnostic information.
- Include reporting and/or recovery information.

```
public class NoMatchingDetailsException
                    extends Exception{

  private String key;

  public NoMatchingDetailsException(String key){
    this.key=key;
  }

  public String getKey(){
    return key;
  }

  public String toString(){
    return "No␣details␣matching"+ key +"found";
  }
}
```

```
public class AddressBook{

    HashMap<Integer, ContactDetails> mycontacts;

    public ContactDetails get(String key)
        throws NoMatchingDetailsException{

        ContactDetails cd = mycontacts.get(key);
        if(cd == null){
            throw new NoMatchingDetailsException("
               Unknown␣Contact");
        }
        return cd;
    }
}
```

```
public ContactDetails getDetails (String key){

    if(key == null){
        throw new NullPointerException("null␣key␣in␣
            getDetails");
    }

    if(key.trim().length() == 0){
        throw new IllegalArgumentException("Empty␣key␣
            passed␣to␣getDetails");
    }

    try{
        return book.get(key);
    }
    catch(NoMatchingDetailsException e){
        // recovery code
    }
}
```

Assertions

Assertions

- Used for internal consistency checks.
    - e.g. object state following mutation.
- Used during development and normally removed in production version.
    - e.g. via a compile-time option.

The Java Assertion Statement

- Two forms available:
- assert boolean−expression
- assert boolean−expression : expression2
- The boolean-expression expresses something that should be true at this point.
- An AssertionError is thrown if the assertion is false.
- The second expression (expression2) is a detailed error message that is passed to the AssertionError (and then to the user).

```
//Assert Statement
public void removeDetails(String key){
  if(key == null){
    throw new IllegalArgumentException (".␣.␣.␣") ;
  }
  if (keyInUse(key) ) {
    ContactDetails details = book.get(key);
    book.remove(details.getName() ) ;
    book.remove(details.getPhone() ) ;
    numberOfEntries --;
  }
  assert !keyInUse(key);
  assert consistentSize () :
       "Inconsistent␣book␣size␣in␣removeDetails";
}
```

Guidelines for Assertions

- They are not an alternative to throwing exceptions.
- Use for internal checks.
- Remove from production code.
- Do not include normal functionality:

```
//Incorrect Use
assert book.remove(name) != null;
```

- In order to use assertions in your pre-production code...
- ... compile using the - -source 1.4 switch
- ... run using the - -ea switch

```
$ javac --source 1.4 MyClass.java

$ java --ea MyClass
```

- For more information, see the Java tutorial

http://docs.oracle.com/javase/8/docs/technotes/guides/language/assert.html

Error Recovery

Error Recovery

- Clients should take note of error notifications.
- Check return values.
- Do not ignore exceptions.
- Include code to attempt recovery.
- Will often require a loop.

```
boolean successful=false;
int attempts = 0;

while (!successful && attempts<MAX_ATTEMPTS) {
  try {
    addressbook.saveToFile(filename);
    successful = true ;
  }
  catch(IOException e) {
    System.out.println("Error␣saving ␣" + filename);
    attempts ++;
    if(attempts < MAX_ATTEMPTS) {
      filename = an alternative file name;
    }
  }
}

if(!successful){
  Report the problem and give up ;
}
```

Error Avoidance

- Clients can often use server query methods to avoid errors.
  - More robust clients mean servers can be more trusting.
  - Unchecked exceptions can be used.
  - Simplifies client logic.
- May increase client-server coupling.

Avoiding An Exception

```
// use the correct method to put details
// in the address book.
if (book.keyInUse( details .getName() ||
      book.keyInUse(details.getPhone()){
  book.changeDetails(details);
}
else{
  book.addDetails(details);
}
```

Review

- Runtime errors arise for many reasons.
    - An inappropriate client call to a server object.
    - A server unable to fulfil a request.
    - Programming error in client and/or server.
- Runtime errors often lead to program failure.
- Defensive programming anticipates errors in both client and server.
- Exceptions provide a reporting and recovery mechanism.

- Input/Output Errors
- Input-output is particularly error-prone.
- It involves interaction with the external environment.
- The java.io package supports input-output.
- java.io.IOException is a checked exception.