5COSC019W – Object Oriented Programming
Week 09

Dr. Barbara Villarini

b.villarini@westminster.ac.uk

Summary

- Introduction to Error and error handling
- Exception
- `try` and `catch` statements
- Multiple `catch` statements
- Using `finally`
- Declaring and Throwing Exceptions
- Creating custom Exceptions
- Testing

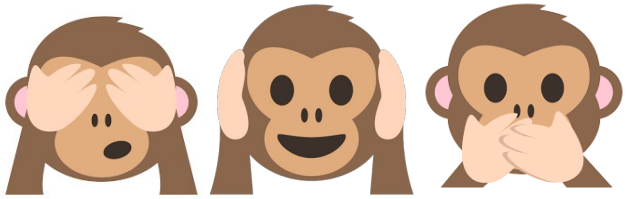
Introduction

- It is extremely rare to write perfect code the first time even though user have high expectation for the code we produce.
- Keep in mind: Users will use our programs in unexpected ways.
- Due to design errors or coding errors, our programs may fail in unexpected ways during execution
- It is our responsibility to produce quality code that does not fail unexpectedly.

Errors and Error Handling

- Some typical causes of errors:
  - Memory errors (i.e. memory incorrectly allocated, memory leaks, “null pointer”) → C++
  - File system errors (i.e. disk is full, disk has been removed)
  - Network errors (i.e. network is down, URL does not exist)
  - Calculation errors (i.e. divide by 0)
  - Array errors (i.e. accessing element –1)
  - Conversion errors (i.e. convert ‘q’ to a number)
  - Can you think of some others?

Possible strategies



- Ignore the problem – not good Idea!!!

- Check for potential problems and abort the program when you find the problem – no good solution!

➡ Check for potential problems, catch the mistake, and attempt to fix the problem.

- Throw an exception. – This is the preferred way to handle the situation

Exceptions

- Most of the OO languages provides features called exceptions.
- Exception is an indication of problem during execution.
- Uses of exception handling
  - Process exceptions from program components
  - Handle exceptions in a uniform manner in large projects
  - Remove error-handling code from “main line” of execution
- In Java, C++, C#, Objective-C and Visual Basic, exceptions are handled by the keywords **`try`**, **`catch`** and **`throw`**.

Handling Exception with Try/Catch block

- Exception handling mechanism:

➢ Find the problem (try block : Hit the exception)

➢ Inform that an error has occurred (Throw the exception)

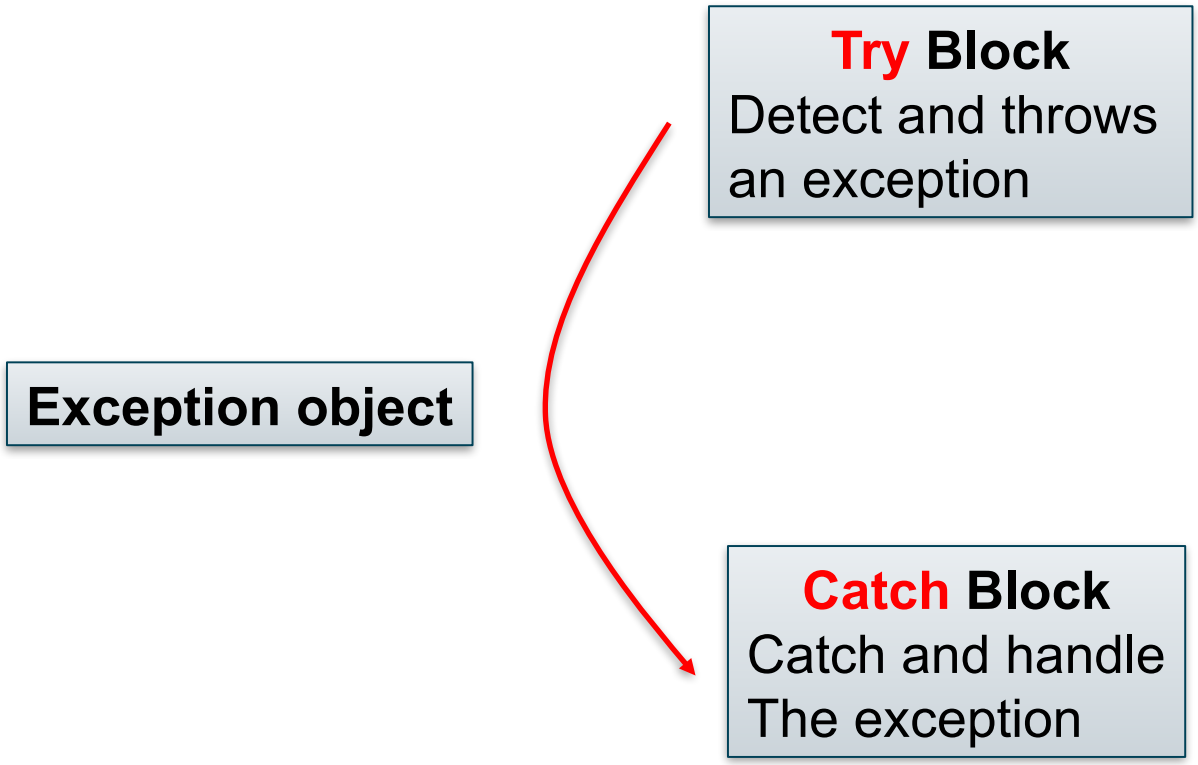
➢ Receive the error information (Catch the exception)

➢ Take corrective action (handle the exception)

Exception handling mechanism

It is basically built upon three keywords

- Try
- Throw
- Catch



Using try and catch

- A structure for a Java try/catch block is the follow:

```
try {
        // possible nasty code
} catch (Exception e){
        // code to handle the exception
}
```

- A `try` followed by any number of `catch` blocks

Example

```
try {
        // possible nasty code
        count = 0;
        count = 5/count;

} catch (ArithmeticException e){

        // code to handle the exception
        System.out.println(e.getMessage());
        count = 1;
}
System.out.println("The Exception is handled");
```

Using multiple catch statements

- You can associate more than one `catch` statement with a `try`.
- Each `catch` must catch a different type of exception.

```
// Use multiple catch statemnts
class ExcDemo4 {
            public static void main(String args[]) {
                        // Here, numer is longer than denom
                        int numer[] = {4, 8, 16, 32, 64, 128, 256, 512};
                        int denom[] = {2, 0, 4, 4, 0, 8};

                        for (int i = 0; i < numer.length; i++){
                                    try{
                                        System.out.println(numer[i] + " / " +
                                                                    denom[i] + " is " +
                                                                    numer[i]/ denom[i]);
                                    }
                                    catch (ArithmeticException exc){
                                                // Catch the exception
                                                System.out.println ("Can't divide by zero!");
                                    }
                                    catch (ArrayIndexOutOfBoundException exc){
                                                // Catch the exception
                                                System.out.println ("No matching element found.");
                                    }
                        }
            }
}
```

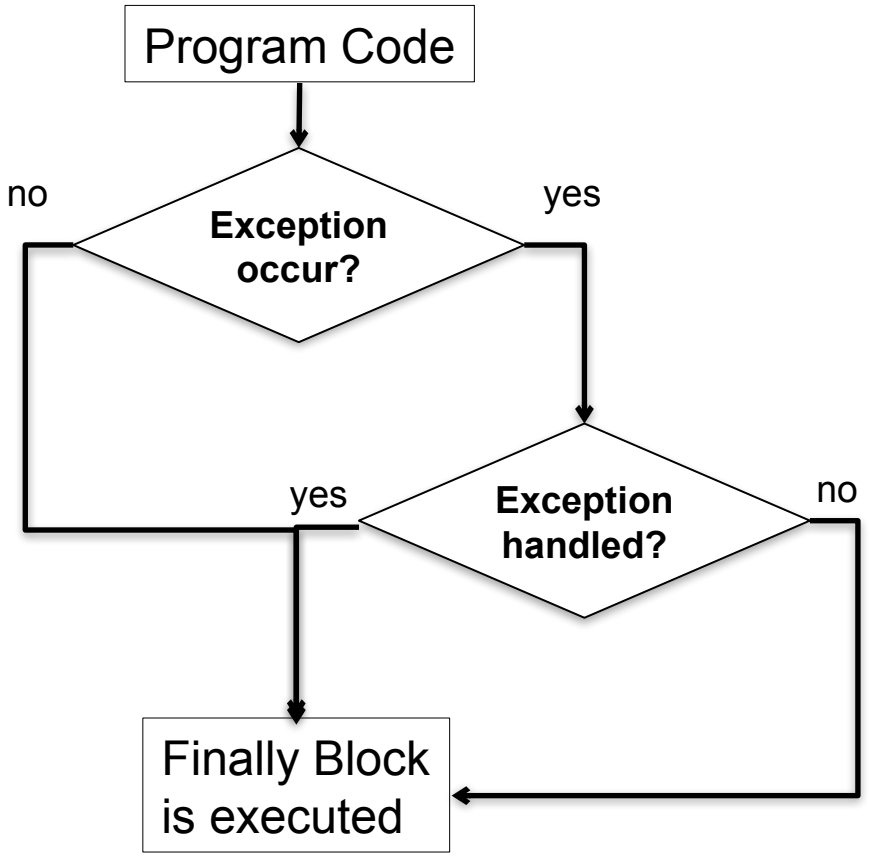
Output

```
4 / 2 is 2
Can't divide by zero!
16 / 4 is 4
32 / 4 is 8
Can't divide by zero!
128 / 8 is 16
No matching element found.
No matching element found.
```

Using `Finally` in Java

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

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



Throwable class in Java

- When an exception takes place, the JVM creates an exception object to identify the type of exception that occurred

- The Throwable class is the super class of all error and exception types generated by the JVM or java programs

- Three Throwable subclass categories are possible:
  - Error (extends Throwable): Serious error that is not usually recoverable.
  - Exceptions (extends Throwable): Error that must be caught and recovered from.
  - Runtime Exceptions (extends Exception) : Error that may be caught if desired.

Checked and Unchecked Exception

- Unchecked Exceptions:
  - They are subclasses of Runtime Exception.
  - *The compiler does not check for these exceptions*, and so a method is not obliged to establish a policy for the unchecked exceptions thrown by its implementation

- Checked Exceptions:
  - They are subclasses of Exception.
  - A checked exception must be caught somewhere in the code, otherwise, it will not compile. So, a method is obliged to establish a policy for all checked exceptions thrown by its implementation.
  - They represent conditions that are reasonably expected to occur, typically representing invalid conditions in areas outside the immediate control of the program (invalid user input, etc).

Catching Subclass Exception

- A `catch` clause for a superclass will also match any of its subclass.
- Since Superclass of all exceptions is Throwable, if you want to catch all the possible exception you can `catch` Throwable.
- If you want to catch exceptions of both subclass type and superclass type, put the subclass first in the catch sequence.

Example

```
// Use multiple catch statements
class Example2 {
        public static void main(String args[]) {
                // Here, numer is longer than denom
                int numer[] = {4, 8, 16, 32, 64, 128, 256, 512};
                int denom[] = {2, 0, 4, 4, 0, 8};

                for (int i = 0; i < numer.length; i++){
                        try{
                            System.out.println(numer[i] + " / " +
                                               denom[i] + " is " +
                                               numer[i]/ denom[i]);
                        }
                        catch (ArrayIndexOutOfBoundException exc){
                                // Catch the exception
                                System.out.println ("No matching element
found.");
                        }
                        catch (Throwable exc){
                                // Catch the exception
                                System.out.println ("Some Exception
Occurred");
                        }

                }
        }
}
```

Output

```
4 / 2 is 2
Some exception occurred.
16 / 4 is 4
32 / 4 is 8
Some exception occurred.
128 / 8 is 16
No matching element found.
No matching element found.
```

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`() to display the stack trace.
  - And a few other methods.

- The Subclasses that extend Throwable can add further variables and methods.

Some Java Built-in exceptions

- ArithmeticException
- ArrayIndexOutOfBounds
- ArrayStoreException
- ClassCastException
- IllegalArgumentException
- IllegalStateException
- IllegalThreadStateException
- IndexOutOfBoundException
- StringIndexOutOfBound Exception

Throwing an Exception

- 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).

- The general form is

```
throw exceptOb;
```

- Where *exceptObj* must be an object of an exception class derived from Throwable

Example

```
public static void parseInt(String s, int num) Throws Exception
{
        if (s == null)
        {
                throw new Exception("null");
        }

        if (num < 0) {
                throw new Exception(”negative number");
        }
}
```

Declaration of throws!

Throw Exception object

Example throwing exceptions

```
// Manually throw an exception
class ThrowExample {
        public static void main(String args[]) {
                try{
                        System.out.println("Before throw.");
                        throw new ArithmenticException();
                }
                catch (ArithmenticException exc){
                                // Catch the exception
                                System.out.println ("Exception caught.");
                }
                System.out.println("After try/catch block.");
        }
}
```

Runtime exception,
You don't need to declare

Output:

```
Before throw.
Exception caught.
After try/catch block.
```

Creating Exception Subclasses

- You can create your own exceptions.
- Just define a subclass of **Exception**.
- The **Exception** class does not define any methods of its own, it inherits those methods defined by **Throwable**.

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 Except

Example – Create an exception

```
// Create an Exception
class NonIntResultException extends Exception{
        int n;
        int d;

        NonIntResultException (int i, int j) {
                n = i;
                d = j;
        }

        public String toString(){
                return ”Result of “ + n + “ / “ + d + “ is non-
integer”;
        }
}
```

Example using a Custom Exception

```
class CustomExceptionDemo{
        public static void main(String args[]) {

                // Here, numer is longer than denom
                int numer[] = {4, 8, 15, 32, 64, 128, 256, 512};
                int denom[] = {2, 0, 4, 4, 0, 8};

                for (int i = 0; i < numer.length; i++){
                     try {
                       if(numer[i]%den[i] != 0)
                          throw new NonIntResultException(numer[i], demom[i]);

                          System.out.println(numer[i] + " / " +
                                                  denom[i] + " is " +
                                                  numer[i]/ denom[i]);
                      }
                      catch (ArithmeticException exc){
                                    // Catch the exception
                                    System.out.println ("Can't divide by zero!");
                      }
                      catch (NonIntResultException exc){
                                    // Catch the exception
                                    System.out.println(exc);

                      catch (ArrayIndexOutOfBoundException exc){
                                    // Catch the exception
                                    System.out.println ("No matching element found.");
                      }
                }
        }}
```

Output

```
4 / 2 is 2
Can’t divide by zero!
Result  of 15 / 4 is non-integer.
32 / 4 is 8
Can’t divide by zero!
128 / 8 is 16
No matching element found.
No matching element found.
```

TESTING

Perfection or Lack of it

- Do your programs work perfectly?
- Are you perfect?

NO!

- No program is perfect.
- Programs will have errors.
- Often see quotes like:

"On average program code has 10 errors per 1000 lines..."

How can we improve our code?

- Validation:
  - "Are we developing the right system?"
  - We should test the behavior against requirements.

- Verification:
  - "Are we building the system right?"
  - We should test the code.

Testing

- Testing will help to find bugs, by actually running the code.
- Testing *cannot* show your program will always work properly – Remember that also the testing can have some bugs!
- But it can remove sufficient bugs to make your program “good enough”.
- Testing allows you to gain confidence in your code.

Black and white box testing

What is the difference between black- and white-box testing?

- **black-box** (procedural) **test**: Written without knowledge of how the class under test is implemented.
  - focuses on input/output of each component or call

- **white-box** (structural) **test**: Written with knowledge of the implementation of the code under test.
  - focuses on internal states of objects and code
  - focuses on trying to cover all code paths/statements
  - requires internal knowledge of the component to craft input
    - example: knowing that the internal data structure for a spreadsheet uses 256 rows/columns, test with 255 or 257

Black box testing and Tests plan

- black-box is based on requirements and functionality, not code
- tester may have actually seen the code before ("gray box")
  - but doesn't look at it while constructing the tests
- often done from the end user perspective
- emphasis on parameters, inputs/outputs (and their validity)

Test Scenario Description	Test Case Description	Test Steps	Pre-Conditions	Expected Result	Test Data	Status
Verify the login Security	Check user login (for an existing account) with invalid email and password.	Enter invalid email and password while initially logging in to the app, then click on the login button.	User must have the app open. user must enter incorrect details.	A text box to pop up asking the user to re-enter a correct/registered email or password.	email: invalidEmail password: invalidpassword	Passed
Verify the login Security	Check user login with valid Data.	Enter valid email and password then click on login button to gain access to the app.	User must have the app open. user must enter correct details.	The application to log into the account as they press the login button.	email: Validname password: validpassword	Passed
Verify the login Security	Check what happens when the login button is pressed without any information.	Press the login button without entering any data into the text fields for email and password.	User must have the app open.	Text to appear and the email and password text field to highlight in red letting the user know the field is empty.	email: password:	Passed
Verify the login Security	Check how many times an incorrect password has been entered.	Enter an invalid password three times.	User must have an existing account User must have the app open. User must enter incorrect password.	After the user enters the password 3 times, a message box will appear asking the user to contact the teacher or the admin to get their account back	password: invalidpassword(x3)	Failed
Check app Functionality	Check if the application lets the user select the context of their conversation.	Display a screen with all the possible contexts of a conversation in the form of buttons.	User must have an existing account User must be logged in. User must have the app open.	Display the conversation for the selected context.		Passed
Check app Functionality and usability	If the user clicks on a tricky word, does the translation box show up?	User clicks on the highlighted word (tricky word)	User must have an existing account User must have logged in. User must have the app open. User must have selected the language, language level, context and subcontext.	Display a pop up box, with tricky words translated in the chosen language.		Failed
Check app Functionality	When the user completes a level, does the progress bar increase.	user completes a conversation at their selected level and then user clicks on the finish level button.	User must have an existing account. User must have logged in User must have the app open.	The progress bar increases by a set amount (depending on the number of levels completed).		Passed
Check app Functionality	When the user signs in, is the progress from the previous session saved?	User enters their login details and then clicks on the login button	User must have an existing account User must have logged in User must have started on a task previously User must have the app open.	The levels completed from a previous session should be greyed out and progress bar should be the same from the last session.		Failed
		The user has to complete a conversation at their chosen	User must have an existing account. User must have logged in.		Pick a language and then	

White Box testing

- "static testing"
  - code walkthroughs, inspections, code reviews
  - static analysis tools
    - CheckStyle http://checkstyle.sourceforge.net/
    - FindBugs http://findbugs.sourceforge.net/
  - code complexity analysis tools
    - PMD, CheckStyle, etc.

- Unit testing: piece of code written by a developer that executes a specific functionality in the code to be tested and asserts a certain behavior or state.
- The percentage of code which is tested by unit tests is typically called test coverage.

Testing and Proof

- To prove something we must show: $\forall x \bullet P(x)$
- This implies we have to explore every possible state a program can be in.
- But how can we explore all the possible state???
- Let's consider a method that compute the sqrt of a number…
- To test if it works correctly we should try all the possible float number, which are $2^{64} \cong 10^{19}$!!!!
- That means we can take ages!!!

Running tests

- Construct the test dataset : select representatives from each of the class
- Create a test harness — a program to call sqrt with the elements of the test dataset.
- Run the program and compare the results with what was expected (which you need to work out some other way!).

Example – Very basic approach (not recommended)

```
public void testSqrt() {
System.out.println("sqrt(1.0) = " + Math.sqrt(1.0)) ;
System.out.println("sqrt(4.0) = " + Math. sqrt(4.0)) ;
System.out.println("sqrt(6.0) = " + Math. sqrt(6.0)) ;
System.out.println("sqrt(10.0) = " + Math. sqrt(10.0)) ;
System.out.println("sqrt(100.0) = " + Math. sqrt(100.0)) ;
System.out.println("sqrt(1000.0) = " + Math. sqrt(1000.0)) ;
System.out.println("sqrt(0.0) = " + Math. sqrt(0.0)) ;
System.out.println("sqrt(-1.0) = " + Math. sqrt(-1.0)) ;
// etc...
}
```

Why Not very good approach?

- This is quickly going to get boring and error prone. – Manual checking process.
  – OK for 10 tests,
  – Tedious for 100 tests,
  – Mind-numbing for 1000 tests.
  – Mistakes will be made.

- Need an automated approach.

➢ Write a test program that reads data from data structure or file.

➢ Get program to run tests and check the results

Example - testing

Max error allowed

```
public void testSqrt(double input, double expected, double delta) {

        double sqrtValue = sqrt(input);
        double err = Math.abs((sqrtValue - expected));

        if (err > delta) {
                System.out.print("Invalid result for sqrt("+input+"),");
                System.out.print(" expected: " + expected);
                System.out.println(", got: " + sqrtValue);
        }
}
```

Example

```
public void run() {
        double[][] d1 = new double[][]
        {{1.0,1.0}, {2.0,1.4142135}, {3.0,1.7320508},
{10.0,3.1622776}, {100.0,10.0}, {0.1,0.316227}, {0.0,0.0},  {-
1.0,Double.NaN} };

testSqrtDataSet(d1);

}

public void testSqrtDataSet(double[][] data) {
        for (int i = 0 ; i < data.length ; i++) {
                testSqrt(data[i][0],data[i][1],0.00001);
        }
}
```

JUnit

- JUnit is a simple, free and open source framework to write repeatable unit tests using Java.
- It is a regression-testing framework, which allows the developer to write code faster with high quality.
- JUnit tests increase the stability of the software.
- The main philosophy behind this testing framework is to make coding and testing move hand in hand.

A case study


```
public class BankAccount {

    private double balance;
    private String accountNumber;

  public BankAccount(double initialBalance){
        balance  = initialBalance;

    }
  public void deposit(double amount){
        //code
        balance += amount;
    }
  public void withdraw(double amount){
        //code
        balance -= amount;
    }
public double getBalance(){
        //code
        return balance;
    }
    public void close(){
        //here
        balance = 0;
    }
}
```

How to write a test case

1. Creates the objects we will interact with during the test. This testing context is commonly referred to as a test's *fixture*. All we need for the testBankAccount are some BankAccount objects.

2. Exercises the objects in the fixture.

3. Verifies the result

```
public class BankAccountTest extends TestCase {
//…
   public void testDeposit() {

        BankAccount account = new BankAccount (50);  // (1)

             a1.deposit(50)    // (2)

       BankAccount accExpected = new BankAccount (100);

        Assert.assertTrue(accExpected.equals(a1));     // (3)
   }
}
```

Assert

- assertEquals(expected, actual)
- assertEquals(message, expected, actual)
- assertEquals(expected, actual, delta)
- assertEquals(message, expected, actual, delta)
- assertFalse(condition)
- assertFalse(message, condition)
- Assert(Not)Null(object)
- Assert(Not)Null(message, object)
- Assert(Not)Same(expected, actual)
- Assert(Not)Same(message, expected, actual)
- assertTrue(condition)
- assertTrue(message, condition)

Structure

- setUp()
  Storing the fixture's objects in instance variables of your *TestCase* subclass and initialize them by overriding the setUp method

- tearDown()
  Releasing the fixture's

- run()
  Defining how to run an individual *test case.*
  Defining how to run a *test suite.*

- testCase()

Well-structured Assertions

```
public class BankAccountTest {
    @Test
    public void depositTest1() {
        BankAccount account = new BankAccount(50);
        account.deposit(50);

        assertEquals(100, account.getBalance(), 0.001);  }
```

The expected value has to be in the left

```
    public void depositTest2() {
        BankAccount account = new BankAccount(50);
        account.deposit(-50);

        assertEquals(50, account.getBalance(), 0.001);  }
```

Tips for testing

- You cannot test every possible input, parameter value, etc.
  - So you must think of a limited set of tests likely to expose bugs.
- Think about boundary cases
  - positive; zero; negative numbers
  - right at the edge of an array or collection's size
- Think about empty cases and error cases
  - 0, -1, null; an empty list or array
- test behavior in combination
  - maybe `add` usually works, but fails after you call `remove`
  - make multiple calls; maybe `size` fails the second time only

Recap

- Exception handling to make your code more robust
- Test to find errors.
- Use a test harness program.
  - Let it do the repetitive hard work.
- Do enough tests to be confident in your code.